LEVEL II

2

AFAL-TR-77-245

AD A056254

DAIS PROCESSOR INSTRUCTION SET EXTENSION STUDY

WESTINGHOUSE ELECTRIC CORPORATION
SYSTEMS DEVELOPMENT DIVISION
BALTIMORE, MARYLAND 21203

AUGUST 1977

TECHNICAL REPORT AFAL-TR-77-245
Final Report for Period 3 May 1976 — 3 August 1977

AIR FORCE AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

This technical report has been reviewed and is approved for publication.

Donald Moon
DONALD MOON, Project Engineer

Mark Michael
MARK MICHAEL, Technical Manager

FOR THE COMMANDER

[illegible]
RICHARD W. SMITH, Lt Col, USAF
Acting Chief
System Avionics Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 **REPORT DOCUMENTATION PAGE**	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 AFAL-TR-77-245 ✓ / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 DAIS PROCESSOR INSTRUCTION SET EXTENSION STUDY	5. TYPE OF REPORT & PERIOD COVERED 9 Final Report, 3 May 76 - 3 Aug 77
	6. PERFORMING ORG. REPORT NUMBER 14 77-0819
7. AUTHOR(s) 10 L. Gray Miller et al.	8. CONTRACT OR GRANT NUMBER(s) 15 F33615-76-C-1292
9. PERFORMING ORGANIZATION NAME AND ADDRESS Westinghouse Electric Corporation, Systems Development Division, Baltimore, Md. 21203 405 897	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 16 2003-04-17 17 04
11. CONTROLLING OFFICE NAME AND ADDRESS AFAL/AA/AAT WP AFB, Ohio 45433	12. REPORT DATE 11 3 August 1977
	13. NUMBER OF PAGES 160 12 165p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

DAIS Processor	AN/AYK-15 Computer
Computer Family	Software Study
Upwards Compatible	Low Level Machine

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of the DAIS Processor Instruction Set Extension Study Final Report is to document the results of a study program of the AYK-15 digital computer. The first phase of the study developed instruction set changes for the AN/AYK-15 computer to increase its software efficiency. The second phase of the study evaluated the impact of these changes on the AYK-15 computer. The final phase of the study defined a block level design of a low cost avionics computer compatible with the recommended instruction set.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

405 897 [illegible]

PREFACE

This report has documented the results of the DAIS study contract. The results of three areas of work will be detailed throughout the following pages.

First, the results and ensuing recommendations of an instruction set analysis are presented in Section 2. Paramount in this work is the selection of base addressing as the most effective method of achieving greater software efficiency. Indeed, base addressing yields a 30 percent improvement in software efficiency when compared with the current AYK-15 instruction set. Also, new data formats for floating-point number representation were analyzed along with integer and fractional representations for fixed-point numbers.

The conclusions of this software analysis were are presented as a recommended instruction matrix in Table 2. This instruction set is then "subsetted" for the Low Level Machine and presented in Table 3.

Second, the hardware and firmware impact of implementing the instruction set of Table 2 on the current AYK-15 computer is analyzed in section 3. The cost impact of the proposed changes are summarized in Table 7.

Finallv, the instruction set of Table 3 is used to investigate the design of a low-level number of the AYK-15 based computer family. Whenever appropriate, performance is sacrified to achieve a minimum parts count for the Low-Level Machine. During this investigation, floating-point instructions are also incorporated into the LLM design. The results of the design are tabulated and presented in terms of performance (instruction speeds), parts and power.

This study shows the desirability and practicality of generating a family of military computers based upon the present AYK-15. With the modifications outlined in this report, the AYK-15 and the LLM provide a sound basis for developing a family of airborne digital computers.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I
PURPOSE

This document is a final report summarizing all facts and conclusions found and drawn in the course of fulfilling DAIS Study 33615-76-C-1292, often referred to as the "DAIS STUDY." The purpose of the contract has been to establish a modified instruction set for the present DAIS computer (AYK-15) and to select a subset of this instruction set to implement a lower performance, upward compatible computer. This report serves as a basis for the definition of an upward compatible computer family for the Air Force.

A preliminary hardware design of the lower performance computer was then performed and is included in this report.

Finally, the impact of modifying the present DAIS computer (AYK-15) to implement the instruction set modifications was investigated.

1.1 INSTRUCTION SET CHOICE

At the outset, a preliminary instruction set was chosen by the AFAL for Westinghouse's use as a baseline in its analysis to determine an optimal instruction set, from a hardware/firmware viewpoint as well as a programmer's, for the proposed computer family. Paramount in the choice of this instruction set (Appendix A of the original contract SOW) was the need to conserve the actual memory space required to encode operational avionics programs. It was recognized that the best way to implement this saving was to create single-length memory reference instructions (16 bits long) which could generate a 16-bit effective memory address (to reference up to 65K words).

Several new addressing modes were proposed as methods of synthesizing 16-bit memory reference instructions:

a. Register Indirect Addressing

b. Register Indirect With Auto Increment

c. Base Relative Addressing

d. Instruction Counter Relative Addressing

e. Immediate Short Formats

f. Immediate Long Formats

Of these new addressing modes the most significant in terms of software efficiency (defined by AFAL purely in terms of the total number of 16-bit words required to code programs) were determined to be Register Indirect and Base Relative addressing. Since both types are each capable of synthesizing 16-bit memory reference instructions, they were posed as alternatives in the selection of the final instruction set. Their relative strengths were then explored by coding a sample avionics problem supplied by the Air Force in each instruction set (i. e., Register Indirect and Base Addressing).

1.2 HARDWARE MODIFICATIONS TO PRESENT DAIS COMPUTER

After the software analysis of the proposed instruction sets was completed, the task of implementing the addressing modes within the framework of the present DAIS computer was studied. This was undertaken in two ways: first considering only firmware (microcode) changes to the present AYK-15 computer with no hardware changes, and secondly with complete freedom to modify or add to hardware as well as firmware.

At this point, the feasibility of the goal of 30 percent improved software efficiency over the present AYK-15 computer with the new addressing modes was analyzed with respect to hardware/firmware/cost tradeoffs, and a final instruction set chosen.

1.3 DESIGN OF LOW LEVEL DAIS MACHINE (LLM)

Another concern in the choice of the optimal instruction set was the feasibility of subsetting the final set for the less powerful members of the computer family. This subsetting also had to maintain an "upwards compatibility" within the family, meaning all instructions used by the "low level" machines would be contained in the "higher level" machines. This insures that operational software which would run on the low level machine would also run on any of the higher level machines in the family.

Westinghouse and AFAL then chose one such subset of the final instruction set for use in its design of a low-level machine (LLM). Generally, this subset contained all instructions of the final set except the floating point arithmetic and double-precision multiplies and divides, keeping the LLM oriented towards a simple, fixed point, front-end processor. (Subsequently, floating point arithmetic was added to the LLM during the design phase.)

A preliminary hardware design of the LLM was then performed. Paramount in this design was the use of the 2900 family of bipolar LSI logic, which has emerged as a front-runner in the rapidly-expanding technology of the LSI field. As currently supplied by Advanced Micro Devices (AMD), Motorola, and Raytheon, this logic family meets Mil-Spec performance criteria, provides low parts count design with low power consumption, and is reliably available on the market. The AM-2901 four-bit microprocessor slice is also structurally compatible with the MM-5701 (used in the AYK-15), making the LLM design directly applicable to the AYK-15.

The primary difference in the two instruction sets was the two addressing modes. Each set contained a "core" of present DAIS instructions (referred to as "DAIS Baseline").

The AFAL supplied a set of three sample avionics programs (DAIS Benchmarks 1, 2, and 3) which are detailed in the document specification number F44615-75-R-1154. Of the three, BENCHMARK No. 1 was chosen by Westinghouse for coding in the two candidate instruction sets.

BENCHMARK No. 1 was divided into six program segments as follows:

(1) Decision and Control

(2) Arithmetic Computation No. 1 and 2

(3) Arithmetic Computation No. 3

(4) Arithmetic Computation No. 4 and 5

(5) LIMIT Subroutine

(6) HMSANG Subroutine

This partitioning was made both to facilitate documentation and to provide for statistical comparison. It isolated Decision and Control, arithmetic processing, and certain subroutines for individual scrutiny.

The statistical comparison was done in two reference frames. First, the relative software efficiency (as defined in Paragraph 1.1) of the two instruction sets from the coding of Benchmark No. 1 was analyzed. Then the frequency of usage of the non-DAIS Baseline instructions (as defined earlier) of each instruction set in the coding of the program was analyzed. This highlighted the relative "strengths" of the new instructions in each set by pointing out how useful each was in solving the Benchmark problem.

SECTION II

INSTRUCTION SET DEFINITION

2.1 FAMILY CONCEPT (UPWARDS COMPATIBILITY)

If a set of computers, all with varying degrees of processing capabilities, are to be considered a computer "family," there must be a direct interrelation among them. A valid measure of the notion of a computer family is the "upwards compatibility" of the machines. This can be determined directly from whether or not a fully operational program written for a smaller member of the "family" can be run directly on a larger family member with the same results.

To this end, the computer family must be "upwards compatible" in terms of software. An instruction set for the "higher" level members of the family should be conveniently subsettable for the "lower" level family members.

Furthermore, a hardware compatibility must be maintained within the family. A fixed set of machine characteristics should be incorporated in each family member, with extensions added to this basic set for the higher level machines. This is done to insure family integrity in data formats, interrupt service, and the like.

2.2 SOFTWARE EFFICIENCY STUDY OF NEW ADDRESSING MODES

As outlined in Paragraph 1.1 of this report, two candidate instruction sets (base relative and register indirect) were assembled to compare the relative strengths of the register indirect and base register addressing formats. The register indirect instruction set was as defined in Appendix A of contract F33615-76-A-1292. (DAIS Study) The base addressing instruction set used was as defined in the Westinghouse-prepared document entitled DAIS Processor Support Software (specification no. MN255R818).

a. RESULTS OF SOFTWARE ANALYSIS

With the Benchmark completely coded in both the Register Indirect and Base Addressing sets, an algorithm was devised to measure the relative software efficiency of the sets. Using the line numbers associated with the program listings, a numerical equation for computing the number of 16-bit instruction words needed by each program segment was formulated:

$$AN_i = (END-BGN) - CMT$$

$$BN_i = (END-BGN) - CMT$$

Where:

AN	= The number of words (instructions plus literals) required to code in AFAL instruction set.
BN	= As above for Base Register instruction set.
END	= Line number of last line.
BGN	= Line number of first line less one.
CMT	= Number of comment lines.

and i = 1, 2, . . . , 6 corresponding to one of six program segments.

Substituting into these equations yielded the following results:

(1) Decision and Control
$AN_1 = (207-19) - 14 = 174$
$BN_1 = (177-19) - 14 = 144$
$$\frac{AN_1}{BN_1} = 1.32$$

(2) Arithmetic Computation No. 1 & 2
$AN_2 = (195-3) - 2 = 190$
$BN_2 = (140-3) - 2 = 135$
$$\frac{AN_2}{BN_2} = 1.41$$

(3) Arithmetic Computation No. 3
$AN_3 = (97-3) - 5 = 89$
$BN_3 = (75-3) - 7 = 65$
$$\frac{AN_3}{BN_3} = 1.34$$

(4) Arithmetic Computation No. 4 & 5
$AN_4 = (172-3) - 3 = 166$
$BN_4 = (144-3) - 3 = 138$
$$\frac{AN_4}{BN_4} = 1.20$$

(5) LIMIT Subroutine
$AN_5 = (39-3) - 1 = 35$
$BN_5 = (38-3) - 1 = 34$
$$\frac{AN_5}{BN_5} = 1.03$$

(6) HMSANG Subroutine

$AN_6 = (98-3) - 2 = 93$ $\frac{AN_6}{BN_6} = 1.24$

$BN_6 = (79-2) - 2 = 75$

TOTALS:

$AN = 747, \; BN = 591$ $\frac{AN}{BN} = 1.26$

These results show the Base Register set of instructions required less program memory than the AFAL set in all six program segments. In total, the AFAL set used 27 percent more program storage than did the Base Register set.

In fact, the AFAL set requires more storage than is reflected in the above figures. Each time a unique address is loaded into the general register used as the "indirect register" an additional location is required. The required word holds the constant whose value is equal to the address in question. For example, on page 51 of the program listing, three locations would be required to save the values loaded into register A4 on lines 59, 63, and 74 respectively. This is different from the base addressing mode, which can address uniquely within its 8-bit displacement range (256 words) with the original base loaded only at the beginning of all references within its boundaries.

b. INSTRUCTION UTILIZATION (AFAL)

Of the 115 AFAL instructions only 17 were used in coding the Benchmark problem. A detailed list follows:

	INSTRUCTION	NO. OF TIMES USED
(1)	RDA	1
(2)	IRS	2
(3)	RDS	1
(4)	IRM	2
(5)	RDM	1
(6)	RDD	1
(7)	RST	8
(8)	IRST	17
(9)	DRST	2
(10)	IDST	9
(11)	IRL	7

(12)	IRDL	2
(13)	JRU	12
(14)	JREQ	7
(15)	JRGT	9
(16)	JRLT	9
(17)	RSB	2

The ratio of instruction types available to instruction types used: 17/115 = 0.15

The ratio of the number of Register Indirect instructions (of all types) used to the total number of instructions required for each of the six program seg..ents are:

(1) 14/117 = 0.12
(2) 13/137 = 0.09
(3) 6/55 = 0.11
(4) 16/111 = 0.14
(5) 12/27 = 0.44
(6) 0/67 = 0.0
TOTAL 61/514 = 0.12

c. INSTRUCTION UTILIZATION (BASE ADDRESSING)

Of the 60 Base Addressing instructions only 18 were used. They were as follows:

	INSTRUCTION	NO. OF TIMES USED
(1)	LB, BR5	55
(2)	STB, BR5	45
(3)	AB, BR5	9
(4)	SBB, BR5	7
(5)	SBB, BR6	1
(6)	MB, BR5	15
(7)	DB, BR5	2
(8)	DLB, BR5	18
(9)	DLB, BR6	1
(10)	DSTB, BR5	20
(11)	DAB, BR5	14
(12)	DSBB, BR5	13
(13)	JCRI, EQ	6
(14)	JCRI, LT	10
(15)	JCRI, GT	8
(16)	JCRD, EQ	1

(17)	JRI	4
(18)	JRD	3

The ratio of instruction types available to instruction types used: 18/60 = 0.30.

The ratio of the number of Base Addressing instructions (of all types) used to the total number of instructions required for each of the six program segments are:

(1) 70/118 = 0.60
(2) 53/120 = 0.44
(3) 29/55 = 0.53
(4) 59/116 = 0.51
(5) 9/25 = 0.36
(6) 13/60 = 0.22
TOTAL 233/494 = 0.47

These ratios show the set of base addressing instructions to be more applicable than the register indirect addressing instructions in a typical avionics problem (such as Benchmark No. 1), both in having more of its instructions applicable in the codings (30 percent to 15 percent) and the overall frequency of their use (48 percent to 18 percent).

Table 1 summarizes the above figures from the comparison of the two instruction sets.

d. CONCLUSIONS OF SOFTWARE ANALYSIS

In terms of software efficiency, it is apparent register indirect addressing is a poorer choice for a short memory reference instruction mode than base register addressing. We can see, from our coding of Benchmark No. 1, a significant savings in memory utilization with the base addressing mode (27 percent less memory space than register indirect).

From the view of utility of instructions, the base register addressing mode again appears to be a better choice. A larger percentage of available base addressing instructions was used (30%) than register indirects (15%), and these instructions were used with over twice the frequency

TABLE 1

INSTRUCTION SET COMPARISON

Program Segment	(1)	(2)	(3)	(4)	(5)	(6)	Total Program
MEM Usage * x (AFAL)/ N (BA)	22%	41%	34%	20%	3%	24%	27%
AFAL Instr Utilization**	38%	9%	11%	14%	44%	0%	18%
BA Instr Utilization **	60%	44%	53%	51%	36%	22%	47%

Notes: * Reflects the percentage by which the AFAL program storage requirement exceeded the Base Addressing program storage storage requirement.

** Reflects the percent of the total instructions which were AFAL (or BA)

77-0819-TA-1

(47 percent to 18 percent) than the register indirects in the solution of Benchmark No. 1. This indicates the base addressing instructions are "richer" in utility for solving typical avionics problems than register indirects, despite being almost half as small a set of instructions (60 to 115). This is also a plus for base addressing, as less instruction order types are necessary for greater utility.

In the process of analyzing the proposed addressing mode changes, many conclusions were reached by the programmers who performed the actual coding. What follows is a summary of their comments about the proposed instruction changes.

For purposes of this discussion, the following instruction word field definitions are used

OT	-	order type code
R_A	-	general register RO, ... R15
R_{EA}	-	general register used to designate an address

R_B	-	general register used as a base address register
D	-	displacement field
N	-	binary number
OCX	-	operation code extension
EXP	-	exponent

2.2.i Register Indirect

Instruction format:

OT	R_A	R_{EA}
16 9	8 5	4 1

Register Indirect addressing is an efficient addressing mode when there are repeated references to the same location. When combined with auto-indexing this advantage is extended to enhance references to adjacent locations. As can be imagined, if a program's data can be structured sequentially the register indirect addressing can provide an increase in software efficiency over double word instructions.

However, if the data base cannot be structured in sequential nature (as will typically be true of all global data blocks), then register indirect addressing will be of very limited use. As an example, consider the two subroutines below. Both subroutines are constrained to use data from a global block of data as is typical of many data structures.

SUBROUTINE A	GLOBAL DATA	SUBROUTINE B
$RO = A/B * C/D$	VAR A VAR B VAR C VAR D	$RO = (\frac{D}{B}) \div C + A$

Structured vs Non-structured Data

Both subroutines are required to perform operations from left to right in order to prevent overflow or underflow. As can be quickly appreciated, Subroutine A is ideally suited for implementing with register indirect addressing since its parameters are stored in the exact sequential order they are needed for computation. However, Subroutine B requires a different ordering of the global variables in order to use register indirect addressing. Of course, some compromise of the sequence of the four variables may be arrived at to allow both Subroutine A, and Subroutine B, to utilize register indirect addressing of their shared variables. However, as the number of users of the global variables grow, the task of organizing the data in an optium fashion for each subroutine user becomes truly Herculean.

It is primarily for this reason that register indirect addressing is inadequate for the computer family. Additionally, once a program is written, the order of storage of the variables may never be altered without a major rewriting of the program itself. This makes program revision doubly difficult and is certainly not in keeping with good programming practices.

2. 2. 2 Base Relative

Instruction format:

OT	R_B	OFFSET
16 11	10 9	8 1

In the process of arriving at the present set of Base Relative instructions, Westinghouse relied heavily on its experience with the predecessor of DAIS, the Millicomputer. This machine used a similar form of base addressing with an eight-bit displacement.

Although not as convenient for coding as double-word instructions, base addressing has proven effective in reducing the memory required to perform avionics problems. Inherent in the use of base addressing is a careful planning of the data structure in order to take advantage of the limited addressing range. It is for this reason that four base registers were chosen. In a typical problem R4 would be used to access a list of global data. Similarly, R5 would be used to access all local variables while R6 would reference a block of "scratch pad" for computation and intermediate results. The last base register, R7, would then be free.

The main disadvantage to base addressing is the restriction to a single accumulator. This definitely presents problems when compared to multiple register capability. However, the full set of register to register instructions, as well as the double word instructions, are available when it is necessary to perform operations on registers other than RO.

The base addressing instructions are not intended to be used solely in a particular application but rather as a supplement to the normal AYK-15 instructions when memory efficiency is desired. To this end they would be used or disregarded as the particular application dictates.

2.2.3 Immediate Short

Type 1: Instruction format:

OT	R_A	$SD_6 - D_O$
16 13	12 9	8 1

($D_6 - D_O$ is a signed seven-bit integer)

Type 2: Instruction format:

OT	R_A	$D_3 - D_O$
16 9	8 5	4 1

($D_3 - D_O$ is an unsigned, four-bit integer whose sign is determined by a bit in the Order Type code field)

Type 1's format for the immediate short would require 48 order type codes to implement only three types of instructions (Load, Add, & Compare). Since this comprises close to 20 percent of the total number of order types available, their usage would have to be extremely high to justify their inclusion. None of these instructions were appropriate for use in the software analysis performed. This high number of order types is too much to pay for three instructions which could not be used in the programs coded.

Type 2's format requires fewer order type codes (six for the three instructions mentioned above), but again has a similar lack of utility.

The value of an immediate short instruction comes into focus when a large number of calculations are done with small integer constants, such as one, two, and the like. This was not the case in Benchmark No. 1. Further, since short instruction types are the primary goal, the load and add immediate short instructions may be performed with the more general base addressing instructions. (This would require the allocation of a literal in a global data block).

2.2.4 Jump Conditional (IC Relative)

Instruction format:

OT	$SD_6 \ldots\ldots D_0$
16 9	8 1

$IC = IC + (D_6 \ldots\ldots D_0)$

This addressing mode, whose signed displacement allows conditional jumping within 127 locations of the present IC value, is definitely advantageous in increasing software efficiency. In solving the Benchmark problem it was applicable for use in approximately 10 percent of the entire program. It is an ideal short format for program loops and small distance jumps.

2.2.5 Jump to Subroutine (IC Relative)

Instruction format:

OT	R_A	$SD_6 \ldots\ldots\ldots D_0$
16 13	12 9	8 1

It is questionable that subroutines could be located within the range of this instruction with high frequency. Unlike the jump conditional instruction discussed above, most subroutines will not typically be co-located to their calling points in the main program, as illustrated by the Benchmark program. This is not a desirable instruction.

2.2.6 Stack (PSH/POP)

We would agree with AFAL in its recommendation for register to memory stack instructions. Since multiple stacking and unstacking of registers is desirable in many program applications (subroutines, argument passing, interrupt save status, etc.), we would suggest the following formats:

Push instruction:

OT	N	R_A
16 9	8 5	4 1

$\{R_A, \ldots, R_{A+N}\} \rightarrow$ STACK

Pop instruction:

OT	N	R_A
16 9	8 5	4 1

Top N locations on stack $\rightarrow \{R_{A-N}, \ldots, R_A\}$

(R15-N-1) $\rightarrow$ R15

It is assumed that R15 is the implied stack pointer. Therefore, the PSH and POP instructions may be used for handling multiple registers. Of course, if N=0 a single register will be transferred.

The use of the stack as an argument-passing instrument is detailed in Paragraph 2.6, Re-entrant Subroutines, of this report.

2.2.7 Immediate Long Formats

Instruction Format:

OC	RA	OCX	I
16 9	8 5	4 1	16 1

This becomes the format for all immediate long instructions. Each of the 16 possible instructions is distinguished by its code in the 4-bit extended op code field OCX. Using the OCX field as such, eliminates any indexed immediate long instructions.

The advantage of this format comes from the ability to compress all the AYK-15 immediate addressing instructions into a single order type code with unique OCX codes. However, the ability to index the operand is sacrificed.

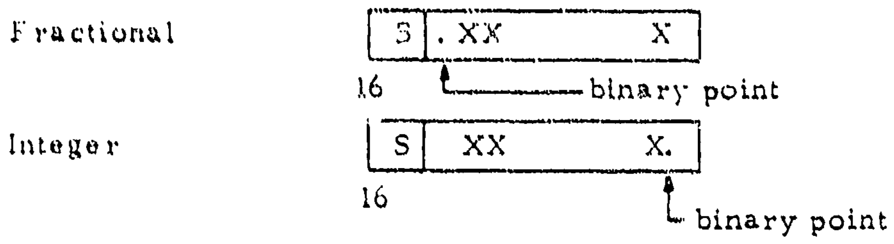
Since immediate addressing is not an important addressing mode (never used) in Benchmark No. 1, it would appear that changes to the immediate addressing structure of the AYK-15 have little impact on software efficiency.

3.3 NEW DATA FORMATS

A suitable set of data formats was to be chosen for the computer family, both for fixed-point and floating-point numbers. Both hardware and software tradeoffs were made for each format.

3.3.1 Fixed-Point Multiply and Divide

A fixed-point number notation must be considered when fixed-point multiply and divide instructions are designed and implemented. The choice for such a notation comes down purely to choosing the position of the binary point. If the binary point is placed at the left end of the 16-bit number, between the sign bit and magnitude bits, the machine is called fractional. If the sign bit is placed at the extreme right end of the number, at the right of the 15 magnitude bits, the machine is considered to be integer:



Since the choice of fractional or integer representation has no significant impact upon the hardware, the choice is truly one of convention. This is illustrated by the widespread use of both conventions by the military computer community:

	MACHINE	MANUFACTURER	NUMBER CONVENTION
(1)	CP-1138 (HARPOON)	Westinghouse	fractional
(2)	AN/YK-15 (DAIS)	Westinghouse	fractional/integer
(3)	SKC-2000	Singer-Kearfott	fractional
(4)	AP-1	IBM	fractional
(5)	4-Pi	IBM	fractional
(6)	AN/UYK-30	Hughes Aircraft	fractional
(7)	AN/UYK-20	Univac	integer

The fractional representation is more common, but again, this is merely a convention. Perhaps the only area where one notation is preferable would be when calculating indices into an array of data. Here, integer representation would be more convenient.

Since AFAL has expressed a preference for integer notation, we would propose that all fixed-point multiplies and divides be made to conform to the integer format.

Also, we would recommend that single precision multiplies return a full 32-bit product. This allows for retention of added significance during single precision computations and is common practice. A summary of the proposed multiply and divide instructions follows.

a. MULTIPLY

(1) 16-bit MPY (M, MR, MI, MIM)
- MPY algorithm is integer
- 32-bit result returned in R_A and $R_A + 1$ (where R_A is even)

(2) 16-bit MPY (MS, MSR, MSI, MSIM)
- MPY algorithm is integer
- 16-bit result returned in R_A

(3) 32-bit MPY (DM, DMR, DMI)
- MPY algorithm is integer
- 32-bit result returned in R_A, $R_A + 1$ (where R_A is even)

b. DIVIDE

(1) 16-bit Divide (D, DR, DI, DIM)
- Divide algorithm is integer
- 32-bit divident in R_A, $(R_A + 1)$ is divided and quotient returned in R_A and remainder is returned in $R_A + 1$ (R_A is even)

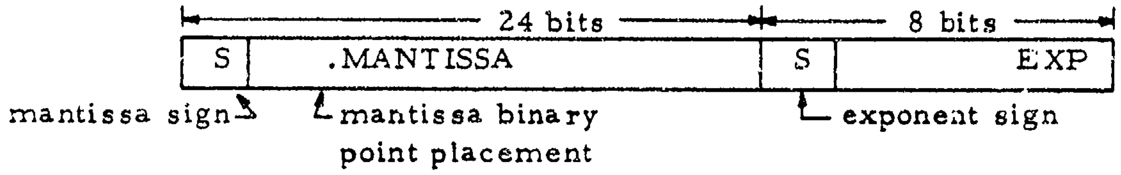
(2) 16-bit Divide (DV, DVR, DVI, DVIM)
- Divide algorithm is integer
- 16-bit dividend in R_A is divided, quotient returned in R_A, remainder returned in $R_A + 1$ (R_A is even)

(3) 32-bit Divide (DD, DDR, DDI)
- Divide algorithm is integer
- 32-bit quotient is returned in R_A and $R_A + 1$, remainder is not saved

2.3.2 Floating Point Format

The choice of a floating point format presents a different type of problem than the fixed-point choice. A floating-point format definitely impacts the amount of hardware necessary for floating-point calculations. Its choice can also affect a utility and readibility to the programmer.

Westinghouse, in its present AYK-15 configuration, has used the following 32-bit format for its single-precision floating-point word:



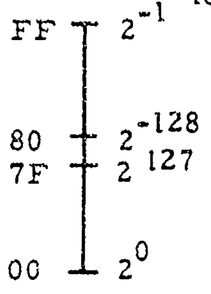
Each bit of the 24-bit mantissa (fractional notation)is as follows:

$$\{ (\text{Sign})\ 2^{-1}\ 2^{-2} \ldots 2^{-23} \}$$

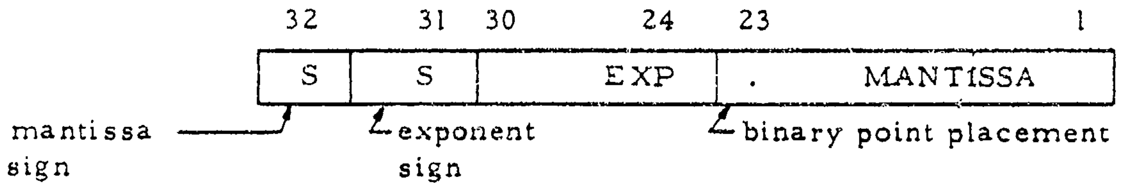
The exponent (8 bits) is in a two's-complement notation, with the following format:

$$\{ (\text{Sign})\ 2^{6}\ 2^{5} \ldots 2^{0} \}$$

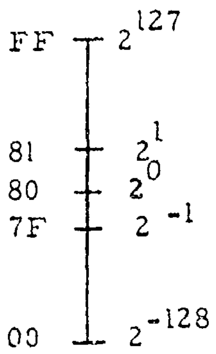
On a sliding scale, from hexidecimal 00_{16} to FF_{16}, the exponent would appear as follows:



The AFAL has suggested a slightly different format for a 32-bit floating point number:



The mantissa, while separate from its sign bit, has the same 24-bit meaning as in the Westinghouse format. The AFAL has suggested, however that the 8-bit exponent be considered as an excess-128 number, meaning the actual exponent value is "offset" by positive 128_{10}. On a hexidecimal sliding scale this looks like:



The two notations give both the same mantissa significance and exponent range ($128 \leq EXP \leq 127$). However, their individual placement in the 32-bit word field turns into a non-trivial difference.

From an aesthetic viewpoint, both formats have pluses. The Westinghouse notation may be slightly more readable, being in the familiar scientific notation order (sign). Mantissa X $2^{(sign)EXP}$. The AFAL notation, on the other hand, has a floating point zero (0×2^{-128}) equivalent in hexidecimal of all zeroes (00000000_{16}) where the Westinghouse format is hex 80 (00000080_{16}).

The individual programmer can also find merits to either convention. In the AFAL format, a relative measure of the sizes of two floating point numbers can be obtained by comparing their integer values, as the major size indicator (exponent) is in the most significant bits of the word and is on a graduated, smallest-to-largest linear scale. This does not "drop out" directly from the Westinghouse format.

The Westinghouse format has the programmer's advantage of being directly accessible to exponent scaling via the machine's byte-mode instructions, as the exponent falls on an eight-bit boundary. The programmer can do a load byte from memory, add, and store byte to accomplish this directly.

These differences pale, however, when compared to the differences in the hardware implemented for floating-point arithmetic. The Westinghouse format makes it simple to "strip" the exponent from the mantissa for processing, and since the exponent is in two's complement notation, a simple addition or subtraction provides the proper new exponent in multiplication or division directly. Exponent over or underflow also falls out directly with no new or extra hardware, because of the four-bit slice structure of the 2901.

The mantissa is also conveniently handled once the exponent is stripped away. The eight bits in the exponent can be directly zeroed out without altering the mantissa value, as they are located in the least significant portion of the 32-bit word. Mantissa overflow in addition or subtraction is also obtainable with no extra hardware.

Floating-point arithmetic becomes much more difficult with the AFAL number representation. The exponent does not fall on an eight-bit boundary, making normal operations on it (adding or subtracting for multiply and divide, or direct number scaling) somewhat more difficult. Also, special hardware must be added to detect exponent overflow or underflow. More hardware and/or firmware is necessary to strip this exponent away for computation.

Mantissa handling is also more difficult. The eight exponent bits can no longer be simply zeroed out, as they are located in the most significant portion of the fraction. Instead, the sign bit must be tested and propagated through these eight bits. This requires yet more special hardware. And still more extra hardware is necessary for mantissa overflow/underflow detection.

The amount of extra hardware necessary for floating-point computations (approximately 15% of the parts count) with the AFAL representation outweighs any advantages it might have from an aesthetic or programmer's view. We recommend the use of the Westinghouse representation on this basis.

2.3.3 Extended Floating-Point Arithmetic

Two extended floating-point formats were also studied. The first was a three-word format, with an eight-bit exponent and 40 bits of mantissa, compared to 24 for the single-precision format. The second was a four-word format, with 56 bits of mantissa.

At approximately three and one-half binary digits per decimal digit of accuracy, roughly seven decimal places are obtainable from the single-precision format, 12 from the three-word extended notation, and 17 from the four-word format.

While the extended floating-point formats do afford an increase in accuracy, there are several points that are well-worth pointing out:

a. When making calculations on extended floating-point numbers, the number of internal registers necessary becomes rather large. A multiply instruction with a 48-bit number requires six registers; for 64 bits, eight registers are necessary. This can severely limit the usage of other available registers for other variables.

b. As the width of the extended format increases, the amount of extra hardware necessary in the EAU (Extended Arithmetic Unit) increases drastically. In jumping from a 24-bit mantissa to a 40-bit length, an extra eight bits must be added to the EAU, which is of 32-bit width. This is an equivalent of 10 to 12 16-pin DIP pack equivalents. And to go to 56-bit mantissas from 40 bits, another 16 bits on top of the eight already mentioned are necessary. At 10 to 12 16-pin packs per eight bits, it would cost 30 to 36 16-pin pack equivalents over the present 32-bit EAU to process the 64-bit format over the 32-bit single precision notation.

c. The added hardware in the EAU would also slow down calculations in the single-precision format. Since the "extended" EAU would "use" all of its hardware even in single-precision mode, several clock times may be wasted in clearing out or sign-extending the upper parts of the registers not used in single precision.

In the light of the above mentioned complications, realizing that the single-precision format is accurate enough for many applications, we do not recommend implementing the extending floating-point formats.

2. 4 CONTEXT SWITCHING

Context (or Mode) switching refers to a major change in the processing "state" assigned to the computer, as would often be encountered at software breakpoints.

The complete "state" of the computer is defined by:

a. The current value of the IC.

b. The Interrupt Mask.

c. The Arithmetic Flags (Overflow, Negative, Zero)

Context switching is accomplished by an orderly replacement of these three quantities by a new set corresponding to the "new state" of the computer. Referring to these three quantities as Program Status Words (PSW's), context switching is performed by "loading the PSW's." Similarly, interrupts may be handled in the same fashion by simply loading in new PSW's to define an interrupt service routine.

2. 4. 1 <u>LPSW Instruction</u>

A new instruction (LPSW) would be added to load the three PSW words (IC, Arithmetic Flags, Interrupt Mask) from successive memory locations pointed to by the effective address. The instruction would be 32 bits long and of the format below.

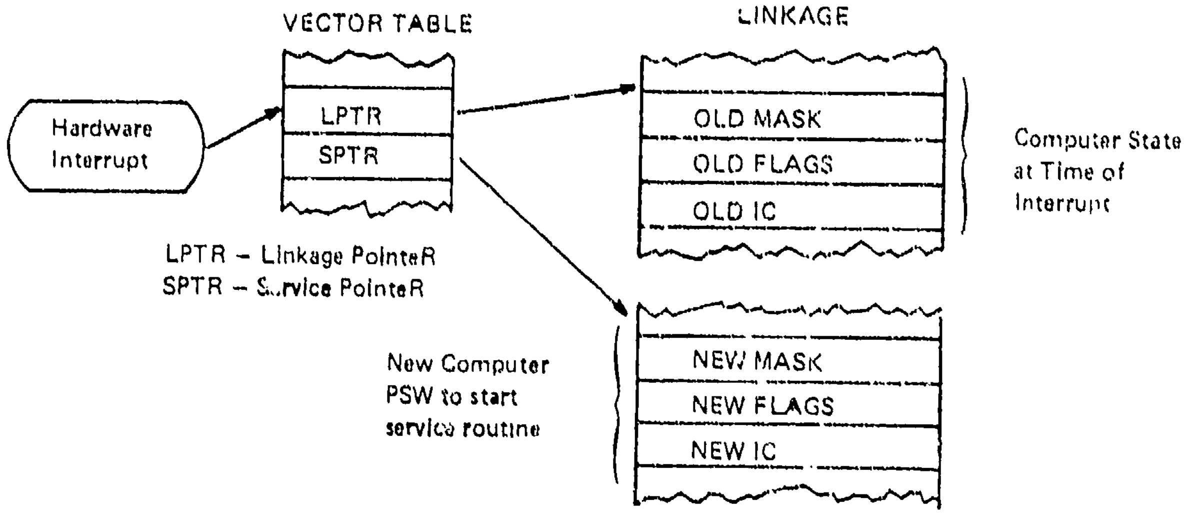
LPSW	X-X	Rx		AF
16 9	8 5	4 1		16 1

Execution of this instruction will then accomplish context switching.

2.4.2 Interrupts

In keeping with the concept of context switching, the hardware interrupt sequence would be altered. The present DAIS machine uses two fixed memory locations to vector each of the 16 possible levels of interrupts. The first memory location would be re-defined as the address of where to store the current PSW's. The second memory location would be similarly re-defined to be the address of the new PSW's to be loaded into the computer. As is customary, this would be accomplished under hardware control.

In schematic form, an interrupt would be handled as follows:



Of course, a return from interrupt would be accomplished by executing the LPSW instruction using the value (LPTR) for an address field.

2.4.3 Priviliged Modes

In data processing type environments, some machine instructions may be reserved for execution by "privileged" users only. This is typically desirable where the user may be inexperienced which requires that the computer's operating system must be protected. However, this has not generally been a problem with military computers due to the high level of refinement enjoyed by an operational program prior to its inclusion in an operational environment.

Nevertheless, should a privileged mode of operation be desirable, it may be entered by a control bit within a PSW word.

2. 4. 4 Multiple Register Sets

The most common scheme adopted by the industry is to offer two sets of registers, thus allowing one to be used for processing interrupts. This obviates the necessity of storing a machine register upon interruption.

Should a second set of working registers be desirable, its selection may be indicated by a bit in a PSW.

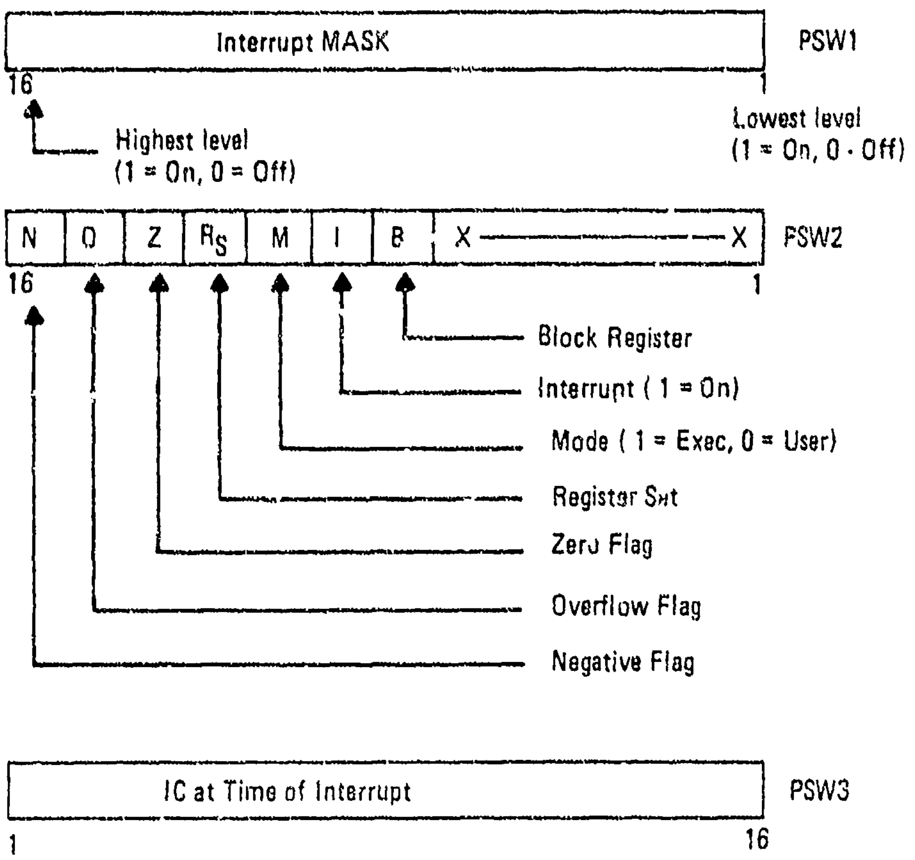
2. 4. 5 Extended Memory Addressing

The present DAIS addressing capability extends to 16 bits, or 65K of memory. This can be extended through the PSW by the inclusion of a block register bit or bits in the word. Each time the PSW is loaded, a block register would also be loaded with the bit value in the PSW. This register would hold the block value until a new PSW is loaded, providing upper bits for memory referencing.

We recommend a one-bit block register, giving up to 130K addressing.

2.4.6 PSW Formats

The three PSW words would be of the format below:



2.4.7 Re-Entrant Subroutines

Subroutines are defined to be "Re-entrant" whenever they may be interrupted by a hardware interrupt and subsequently called prior to their completion of the interrupted computation. Therefore, all intermediate results from an interrupted subroutine must be saved and then restored when the interrupted subroutine is allowed to resume.

If intermediate results are entirely contained within the register set then simply preserving the register set upon interruption is sufficient for implementing re-entrant subroutines. However, if intermediate values are held in scratch memory, then this memory must be reserved at the time of interruption (and not returned for use as common scratch). The collection of information necessary to "re-enter" an interrupted subroutine

(i. e., the intermediate values, etc.) at the point of interruption is said to be "Interrupt Linkage."

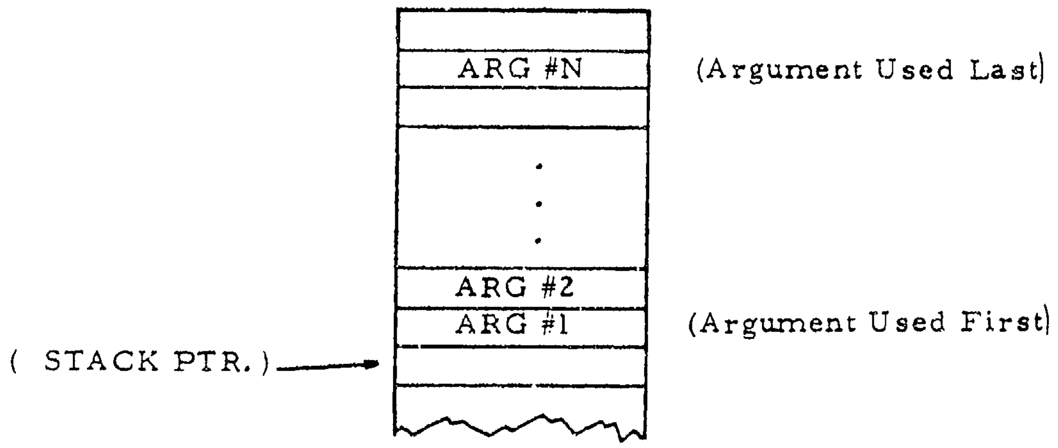
If a re-entrant subroutine is allowed multiple interrupts then multiple sets of interrupt linkage must be preserved.

Not all subroutines need be re-entrant. (In fact, Westinghouse software does not allow re-entrant subroutines due to their aforementioned complexity). However, a generalized scheme for implementing re-entrant subroutines on the present AYK-15 machine will be presented. Also, alternatives to the present implementation will be presented.

2.4.7.1 Subroutine Argument Passing

By convention, arguments will be pushed onto a STACK prior to calling a subroutine. Therefore, if N arguments are passed to a subroutine, the calling program will first push all N arguments onto the stack prior to calling a subroutine. Presumably the arguments will be pushed in the order the subroutine requires their use. Also, the calling program will assign a scratch memory area to the subroutine by passing a starting address to the subroutine as an argument.

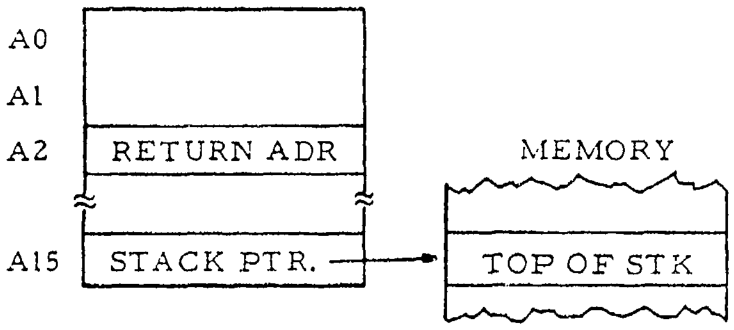
At the time of a subroutine call, the stack will be configured as follows:



2.4.7.2 Subroutine Calls

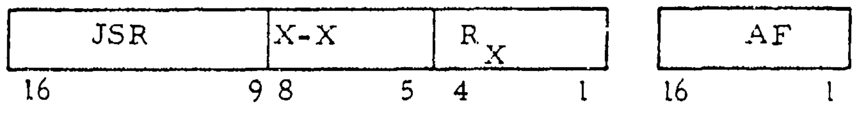
2.4.7.2.1 <u>Present DAIS</u> - Subroutine calls are performed by a jump subroutine (JS) instruction (refer to DAIS Processor Support Software,

p. 124). The return linkage is placed in the register specified by the R1 field of the instruction. As described, this instruction also implements the subroutine return. Therefore, at the beginning of a subroutine, if A2 contains the return linkage, the register set will be as follows:



If nested subroutines are allowed, then A2 must be saved prior to the next call.

2.4.7.2.2 Proposed Change - Alternately, the return linkage may be placed on a STACK so that returns may be accumulated to accommodate re-entrant code. An instruction to call a subroutine of the format below would be necessary.



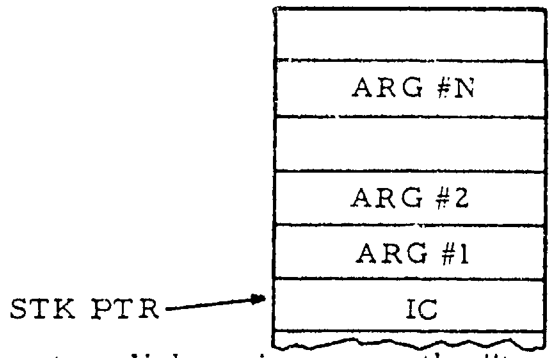
$$IC \rightarrow STK$$
$$(R_X + AF) \rightarrow IC$$

It is assumed that one of the general purpose registers would be an implied stack pointer.

The calling sequence for a subroutine would then be:

```
STK       ARGN
STK       ARG (N-1)
 .
 .
 .
STK       ARG 1
JSR       SRTN
```

At the time of the call the stack would be:



Note that the return linkage is now on the "top of the stack." The subroutine must first "pop the stack" to save the return linkage prior to popping any arguments. Thus it would seem preferable to simply leave the return linkage in a register.

Finally, a RETURN instruction must be added to pop the return linkage into the IC. This, however, can be a short instruction since all addresses are implied. The return instruction would be:

RTRN	X———X
16 9	8 5 4 1

(Top of STK) → IC

Now a complete comparison can be made of the two methods of handling return linkage. Consider the two calling and return sequences shown below:

Present DAIS			Proposed Change			
CALLING PROGRAM			CALLING PROGRAM			
	JS	A2		JSR	SRTN	
SUBROUTINE			SUBROUTINE			
SRTN			SRTN	USTK	TEMP	SAVE LINK
				———		
	———			———		
	———			———		
	———			———		
	———			———		

J	O, A2	RTRN

Total word to Call & Return = 4 Total words to call & return = 5

If we compare the Subroutine Overhead (number of words to link and return from a subroutine) we find that the stacking mechanism requires one more word. Therefore, the two methods seem nearly equivalent in terms of software efficiency.

2. 4. 7. 3 Hardware Implications

Employing a stacking mechanism for subroutine returns requires addition of the RROM as specified in Section 2. 5. 2.

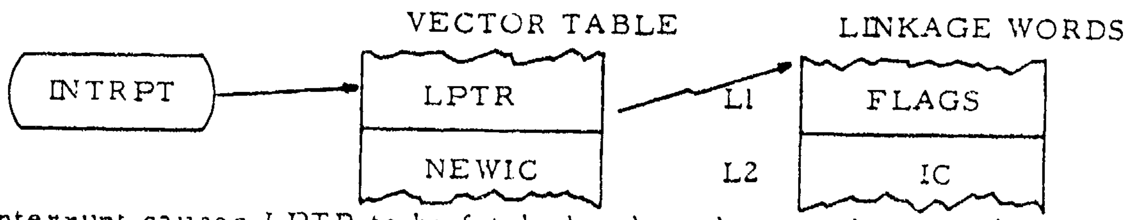
The PSH and POP instructions as defined in Paragraph 2. 2. 6 would require minor hardware modifications to the present AYK-15. Table 4 presents the summary of modifications necessary to the present AYK-15 processor.

Microcode flowcharts for the PSH, POP, and LPSW instructions are presented in Paragraph 3. 3.

2. 4. 7. 4 Interrupt Routines

If re-entrant subroutines are to be allowed, then a complete saving of machine status (arithmetic flags, registers, and IC) is necessary upon receipt of a hardware interrupt. Further, if nested subroutines are to be allowed then stacking of interrupt linkages is desirable.

2. 4. 7. 4. 1 Interrupt Stacking - Present DAIS - Interrupt linkages may be stacked in the present DAIS machine by use of the STK and SM instructions. Recalling the interrupt structure of DAIS,



an interrupt causes LPTR to be fetched and used as a pointer to the linkage words. After the arithmetic flags and incremented IC are stored in the linkage words, the service routine is begun at address NEWIC.

To provide complete linkage stacking the service routine will be:

```
        ' BEGINNING OF INTERRUPT SERVICE
NEWIC       STK  A15, L1                    . STACK FLAGS
            STK  A15, L2                    . STACK IC
            SM   15, 0, A15                 . STACK REGISTERS
            AIM  A15, (17₁₀)                . MOVE STK PTR
            ___
            ___   )
            ___   )  BODY OF SERVICE ROUTINE
            ___   )
            ___

        ' END OF SERVICE ROUTINE
               SIM     A15, (16₁₀)          . MOVE STK PTR
               LM      15, 0, A15           . RESTORE REG.
               USTK    A15, L2
               USTK    A15, L1
               EXS     L1                   . RETURN
        ' END INTERRUPT SERVICE ROUTINE
```

2.5 CONCLUSIONS

2.5.1 Summary of Proposed Changes

2.5.1.1 Utilizing Only Firmware Changes

As can be seen from Tables 4 and 5, the only modifications which can be accommodated on the present DAIS machine with no hardware impact is register indirect addressing. Hence, if this were the only modification made to the present DAIS computer, new microcode could be added to the existing machines (provided some "S-types" were eliminated) to form the nucleus of the computer family.

However, as discussed in section 2, we have been unable to achieve the desired level of software efficiency (30% improvement over present AYK-15) by using only register indirect addressing as an addition to the present DAIS baseline instructions. For this reason we would conclude that firmware changes alone are not sufficient to satisfy the goals of this study.

2.5.1.2 Utilizing Hardware and Firmware Changes

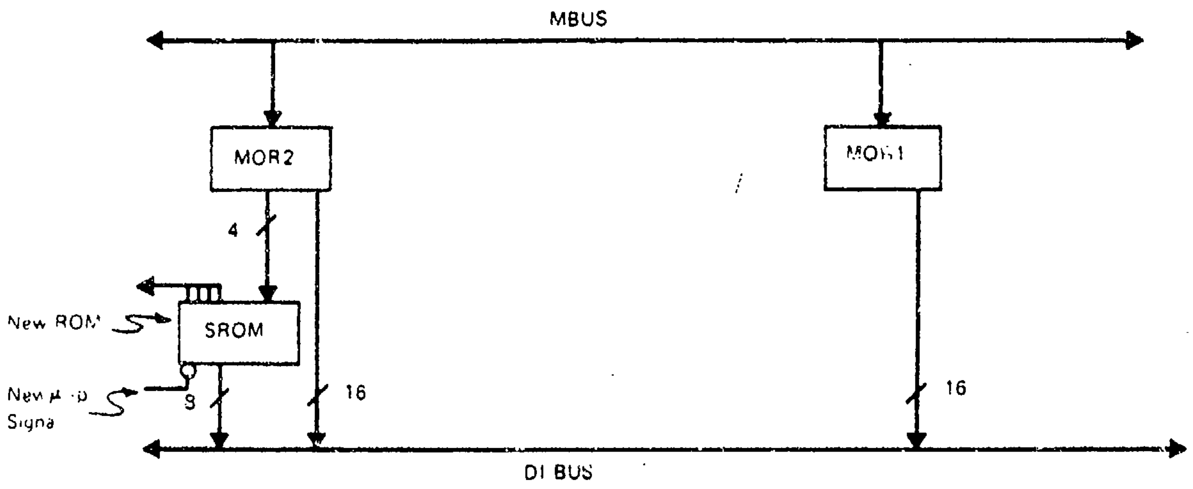
Section 2, illustrated that the desired improvement in software efficiency can be achieved by the addition of base relative addressing. Although requiring minimum additional hardware, the benefits to software efficiency are most dramatic (~36% improvement over present AYK-15). Therefore, we would recommend that the hardware changes listed in Paragraph 2.5.1.1 be incorporated into the present DAIS machine.

These changes would require the alteration of MC1 and MC2, to allow for the addition of the RROM and S-Gates as shown in figures 2 through 3. Also, some minimal backpanel wiring changes would be necessary between MC1 and MC2. Although requiring changes to two printed wiring boards, these changes are, conceptually, of minimal complexity.

Therefore, incorporation of the hardware changes to accommodate base relative addressing, is the only acceptable alternative to achieving the desired increase in software efficiency and should be incorporated into the present AYK-15 machine.

In tables 4 through 7, each case is expressed separately. If multiple cases were to be incorporated, the "costs" in the columns labeled microcode required, hardware required, labor, and parts are not necessarily added. For example, a memory controller card would require new artwork for one change or many changes, and microcode routines would be shared for different changes. If necessary, new microcode storage would be added.

Purpose To translate the RA,RB fields of the 6-Bit Base Relative format to register addresses

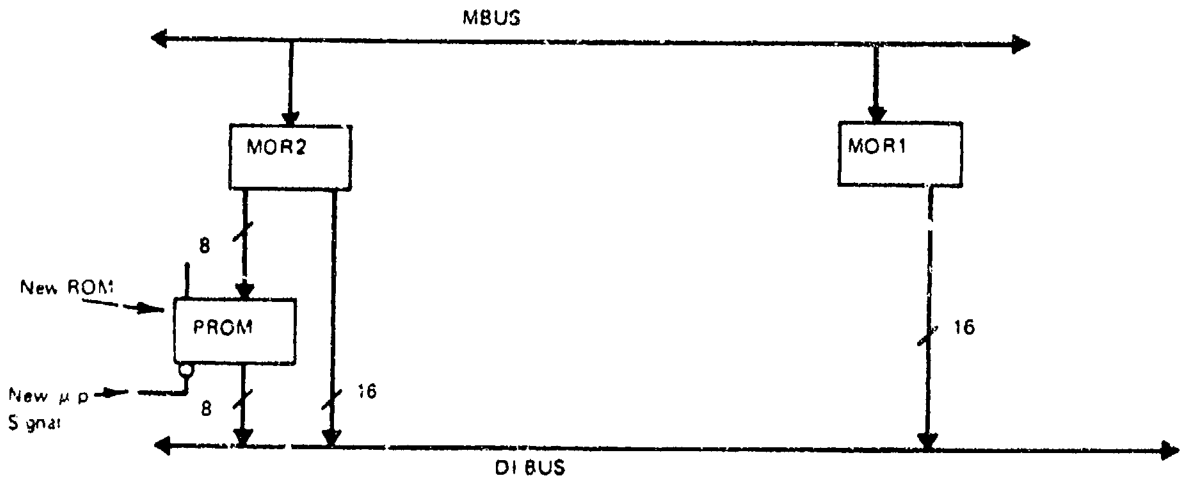


Additional Hardware 2 1 2 16-pin equivalent packs
Changes. MC1, MC2, backpanel

77 0819-VA 2

Figure 1. SROM

Purpose: Used to generate register addresses from the order type code which is contained in the MSB of MOR2.



Additional Hardware 2 1 2 16 pin equivalent packs
Changes MC1, MC2, backpanel

77-0819-VA 3

Figure 2. RROM

Purpose To sign extend an address field for 16 bit arithmetic with the CPU

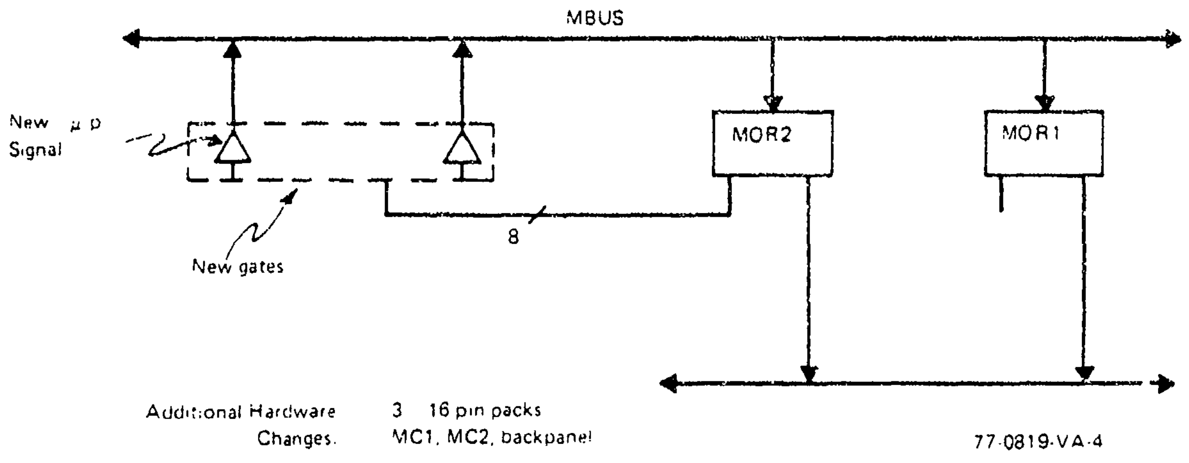

MBUS
New μp Signal
New gates
8
MOR2
MOR1
Additional Hardware 3 16 pin packs
Changes. MC1, MC2, backpanel
77-0819-VA-4

Figure 3 . S-Gates

Purpose To translate the OCX field of immediate long instructions to starting addresses for μcode

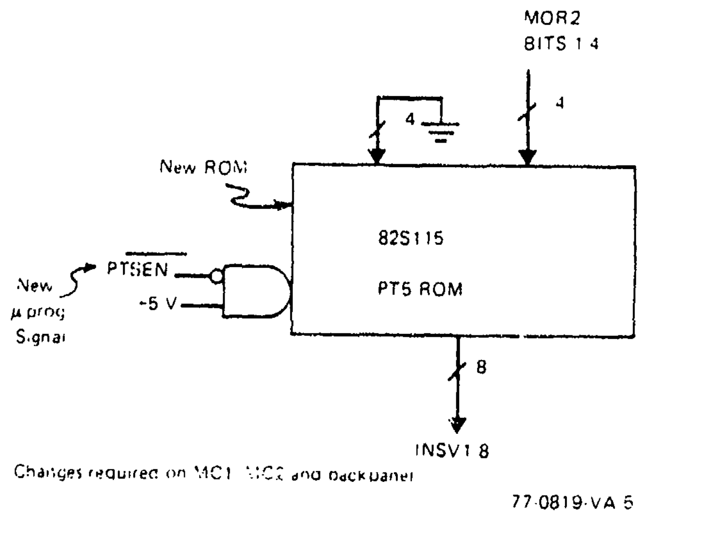


Figure 4 , PT5ROM

2. 5. 2 Final Instruction Set

Table 2 illustrates the final instruction set for the modified DAIS computer as chosen from the findings of this study.

2. 5. 3 Subset for Low Level Machine

When choosing an instruction set for the "low-level machine" of the computer family, we would recommend that a subset of the instructions of Paragraph 2. 5. 2 be chosen. Further, those instructions which required unique hardware to implement should be excluded from this set. This will enable the low-level machine to reach a minimum parts count with the ensuing advantages of low volume, power, and cost.

In keeping with this goal, we would recommend the elimination of the floating-point instructions, as well as the double precision multiplies and divides. Both these instruction types require unique hardware due to their complexity.

The elimination of these instructions would be in keeping with the goal of a low-level machine oriented towards the simple, fixed point, front-end processor.

Table 2 illustrates, in instruction matrix form, the subset of instructions

TABLE 2

RECOMMENDED INSTRUCTION MNEMONICS IN MATRIX FORM

MOST SIGNIFICANT HEX DIGIT OF OPERATION CODE

LEAST SIGNIFICANT HEX DIGIT OF OPERATION CODE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	BASE ADDRESSING					BIT	SHIFT	JUMP	LOAD	STORE	ADD	SUB	MUL	DIV	LOGIC	COMPAR
0	LB BR4	MB BR4	DAB BR4	ORB BR4		SB	SLL	JG	L	ST	A	B	M	D	OR	C
1	LB RB5	MB RB5	DAB RB5	ORB RB5		SBR	SRL	JCI	LR	STZ	AR	SR	MR	DR	ORR	CR
2	LB BF6	MB BR6	DAB BR6	ORB BR6		SBI	SRA	JS	LI	STI	AI	SI	MI	DI	ORI	CI
3	LB BR7	MB BR7	DAB BR7	ORB BR7		RB	SLC	JIX	LIM	STZI	AIM	SIM	MIM	DIM	ORIM	CIM
4	STB BR4	DB BR4	DSBB BR4	XORB BR4		RBR	SRC	JBC	DL	DST	DA	DS	DM	DD	AND	DC
5	STB BR5	DB BR5	DSBB BR5	XORB BR5		RBI	DSLL		DLR	SRM	DAR	DSR	DMR	DDR	ANDR	DCR
6	STB BR5	DB BR6	DSBB BR6	XORB BR5		TB	DSRL	EX	DLI	DSTI	DAI	DSI	DMI	DDI	ANDI	DCI
7	STB BR7	DB BR7	DSBB BR7	XORB BR7		TBR	DSRA	BPI	LUB	STUB	FA	FS	FM	FD	ANDM	FC
8	AB BR4	DLB BR4		JCRI EQ		TBI	DSLC		LLB	STLB	FAR	FSR	FMR	FDR	XOR	FCR
9	AB BR5	DLB BR5		JCRI L.T.			DSRC	CIO	STK	USTK	FAI	FSI	FMI	FDI	XORR	FCI
A	AB BR6	DLB BR6		JCRI G.T.		SVBR	SLR		I	O	IM	DMM	MS	DV	XORI	
B	AB BR7	DLB BR7		JCRD EQ			SAR		PSH	POP	ABS	NEG	MSR	DVR	XORM	
C	SBB BR4	DSTB BR4	ANDB BR4	JCRD L.T		RVBR	SCR	LUBI	FIX	MOV	DABS	DNEG	MSI	DVI	N	
D	SBB BR5	DSTB BR5	ANDB BR5	JCRD G.T.			DSLE	LLBI	FLT	MOVI	FABS	FNEC	MSIM	DVIM	NR	
E	SBB BR6	DSTB BR6	ANDB BR6	JRI		TVBR	DSAR	SUBI	LPSW						NI	
F	SBB BR7	DSTB BR7	ANDB BR7	JRC			DSCR	SLBI	LM	STM					NIM	NOP

77 0819 TA 6

TABLE 3

DAIS FAMILY LOW-LEVEL MACHINE - RECOMMENDED

INSTRUCTION MNEMONICS IN MATRIX FORM

MOST SIGNIFICANT HEX DIGIT OF OPERATION CODE

LEAST SIGNIFICANT HEX DIGIT OF OPERATION CODE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	BASE ADDRESSING					BIT	SHIFT	JUMP	LOAD	STORE	ADD	SUB	MUL	DIV	LOGIC	COMPARE
0	LB BR4	MB BR4	DAB BR4	ORB BR4		SB	SLL	JC	L	ST	A	S	M	D	OR	C
1	LB RB5	MB RB5	DAB RB5	ORB RB5		SBR	SRL	JCI	LR	STZ	AR	SR	MR	DR	ORR	CR
2	LB BR6	MB BR6	DAB BR6	ORB BR6		SBI	SRA	JS	LI	STI	AI	SI	MI	DI	ORI	CI
3	LB BR7	MB BR7	DAB BR7	ORB BR7		RB	SLC	JIX	LIM	STZI	AIM	SIM	MIM	DIM	ORIM	CIM
4	STB BR4	DB BR4	DSBB BR4	XORB BR4		RBR	SRC	JBC	DL	DST	DA	DS			AND	CK
5	STB BR5	DB BR5	DSBB BR5	XORB BR5		RBI	DSLL		DLR	SRM	DAR	DSR			ANDR	CCR
6	STB BR5	DB BR6	DSBB BR6	XORB BR6		TB	DSRL	EX	DLI	DSTI	DAI	DSI			ANDI	CCI
7	STB BR7	DB BR7	DSBB BR7	XORB BR7		TBR	DSRA	BPT	LUB	STUB					ANDM	
8	AB BR4	DLB BR4		JCRI EQ		TBI	DSLC		LLS	STLB					XOR	
9	AB BR5	DLB BR5		JCRI LT			DSRC	CIO	SIX	USTK					XORR	
A	AB BR6	DLB BR6		JCRI GT		SVBR	SLR		I	O	IM	DSSM	MS	DV	XORI	
B	AB BR7	DLB BR7		JCRD EQ			SAR		PSH	POP	ABS	NEG	MSR	DVR	XORM	
C	SBB BR4	DSTB BR4	ANDB BR4	JCRD LT		RVBR	SCR	LUBI		MOV	DABS	DNEG	MSI	DVI	N	
D	SBB BR5	DSTB BR5	ANDB BR5	JCRD GT			DSLR	LLBI		MOVI			MSIM	DVIM	NR	
E	SBB BR6	DSTB BR6	ANDB BR6	JRI		TVBR	DSAR	SUBI	LPSW						NI	
F	SBB BR7	DSTB BR7	ANDB BR7	JRD			DSCR	SLBI	LM	STM					NIM	SOP

77 0829 TA 3

SECTION III

MODIFICATIONS TO PRESENT DAIS

3.1 MICRO-CODE

The hardware and firmware (micro-code) implications of modifying the present DAIS machine to include the new instructions, addressing schemes and floating-point arithmetic formats are presented in tables 4, 5 and 6, respectively.

3.1.1 Instruction Changes

Each instruction option (table 4) and addressing mode (table 5) is evaluated with respect to six parameters.

a. #OT Codes: The number of Order Type Codes required for the instruction or addressing mode.

TABLE 4

NEW INSTRUCTION EVALUATION

Change	OT Codes	Time (μsec)	CPUμp	MC μp	Hard Req'd	Physical Changes
1. PSH	1	(2.8 + 1.4 N)	6	10	PROM RSAV	MC1, MC2 CPU Backpanel
2. POP	1	(3.0 + 1.6 N)	7	9	PROM RSAV	MC1, MC2 CPU Backpanel
3. LPSW	1	3.8	6	15	--	INT Backpanel

77-0819-TA-8

TABLE 5

DAIS STUDY ADDRESSING MODE EVALUATION

(SHEET 1 OF 2)

ADDRESS MODE	# OF CODES	ADD TIME (usec)	CPU # IC	MC # IC	HARD REQ'D	PHYSICAL CHANGED	COMMENTS
1 REGISTER INDIRECT OT | RA | RB 16 9 8 5 4 1	40	2.4/2.0	1	40 (1)	None	-	
2 REGISTER IND WITH AUTO INC OT | RA | RB 16 9 8 5 4 1	15	2.4/2.0	4	6	None	-	
3 IMMEDIATE SHORT (7) OC | RA | I 16 9 8 5 4 1*	5	2.0/2.0	6	7	1 gate pack Backpanel	MC 2 Backpanel	
4 IMMEDIATE SHORT OT | XXXX | S D6 D0 16 13 12 9 8 1	48	2.4/2.0 (2) 2.0/2.0	5 0	7 2	PROM PROM + S-gates	MC1, MC2, Backpanel	
5 JUMP COND IC REL OT | S D0 D6 16 9 8 1	7	- - - -	5 2	8 3	PROM PROM + S-gates	MC1,MC2, Backpanel	
6 JUMP SUB IC REL OT | XXXX | S D6 -- D0 16 13 12 9 8 1	16	- - - -	5 2	7 2	PROM PROM + S-gates	MC1,MC2, Backpanel	
7 IC RELATIVE SHORT (8) OC | D 16 9 8 1*	9	1.4/2.0	7	4	None	PROM S-gates MC1, MC2, Backpanel	Jump or Carry special case WRT Hardware
8 6 BIT BASE REL OT | RA | RB | D5 D4--D0 16 11 10 9 8 7 6 1	7	4.4/2.0 (2) 2.0/2.0	10 1	13 3	SROM SROM + S-gates	MC1,MC2, Backpanel	only for "non S" instruction
9 BASE RELATIVE SHORT OC | BR' | D 16 11 10 9 8 1**	70	2.6/2.0 (4)	20	32	PROM 1 nandgate chip 1 JK FF S-gates	MC1,MC2, Backpanel	
10 IMMEDIATE LONG (5) OC | RA | OCX | I 0 7 8 11 12 15 16 1*	1 for 11 minor OP Codes	1.4/2.0	8 (6)	19 (6)	PT5 ROM	MC1, MC2, Backpanel	Generate two new instr. MIM & SDIM

Circled numbers refer to notes in text section 3.1
* Addressing modes investigated per SOW amendment No. 1
** Addressing mode investigated per SOW amendment No. 2

77 0819 TA 9

TABLE 5

DAIS STUDY ADDRESSING MODE EVALUATION

(SHEET 2 OF 2)

Associated notes and comments for table 5

(1) The present DAIS machine has 60 spare CPU μ program words and 40 spare memory controller μ program words.

(2) It should be noted when evaluating table 5 that some addressing modes have two sets of entries (e.g. Immediate Short). This is because two hardware methods of implementation were evaluated and the results of each presented.

(3) Although the number of memory controller μ program locations required for Register Indirect is very high (40), this could be reduced considerably by a more prudent choice of register indirect instructions. Specifically, whenever an "S type" instruction (as specified in DAIS computer documentation) is required to have a register indirect format, two unique memory controller μ program locations are required. (There are 17 "S type" instructions with register indirect formats as specified by the contract SOW)

(4) The execution times for the Base Relative Short instructions given in table 5 is an average time. The actual times are

a. Single word fetch instructions 2.6 μsec
OP codes 00,01,02,03
08,09,0A,0B
0C,0D,0E,0F
10,11,12,13
14,15,16,17
2C,2D,2E,2F
30,31,32,33
34,35,36,37

b. Single word store instructions 2.8 μsec
OP codes 04,05,06,07

c. Double word fetch instructions 2.6 μsec
OP codes 18,19,1A,1B
20,21,22,23
24,25,26,27

d. Double word store instructions 3.2 μsec
OP codes 1C,1D,1E,1F

e. Jump conditional, relative 2.0 μsec no branch/2.2 μsec branch
OP codes 38,39,3A (Increment)
3C,3D,3E (Decrement)

f. Jump relative 2.0 μsec
OP codes 3B (Increment)
3F (Decrement)

(5) The Immediate Long Instructions format investigated eliminates indexed immediate long instructions. This means the programmer can no longer do

$R5 + 3 \rightarrow R_2$ - LIM R_2, +3, R5

as one instruction, but must now do:

$R5 \rightarrow R_2$ = LR

$R_2 + 3 \rightarrow R_2$ = AIM

Likewise, $R_2 + R5 + 3 \rightarrow R_2$ = AIM R_2, 3, R5

but must also do, $R5 \rightarrow R_2$ = LR

$R_2 + 3 \rightarrow R_2$ = AIM

(6) The instructions MIM (16 x 16 = 31) and SDIM (16 ÷ 16 = 16) are not presently implemented and the microcode necessary is included in this chart.

(7) The range of I is $1 \leq I \leq 16$, therefore the programmer and/or assembler will have to code the following values for I:

I_{10}	BIT VALUES
1	0000
2	0001
15	1110
16	1111

The CPU hardware will add 1 to I and assign the correct sign as designated by the OP code.

(8) The present DAIS machine architecture contains a 4-bit condition status register with one-bit allocated to each of the following conditions:

a. Less than zero, less than (condition)
b. Equal zero, equal (comparison)
c. Greater than zero, greater than (comparison)
d. Overflow, underflow, abnormal, etc

This does not accommodate a jump on carry condition. However, the carry result is available from the carry save flip-flop, and is used during micro-code branch conditions. By specifying a separate op code, new micro-code can be written to generate the desired Jump On Carry instruction. All required hardware exists, only firmware changes are required.

77-0819-TA-10

TABLE 6

FLOATING - POINT INSTRUCTION FORMATS

OPTION	HARDWARE CHANGES			μ PROGRAM CHANGES			
	MC	CPU	EAU	MC	CPU	EAU	COMMENTS
1. Single Precision Format S | E | M 1 8 23	–	Backpanel Wiring	Existing ① +10 add'l Parts	–	80	65	
2. Double-Precision Format 1 S | E | M 1 8 23 M 16	-	Backpanel Wiring	Existing ② + 34 add'l parts	40 ② New Loc	131	65	Impacts speed of other instructions
3. Double-Precision Format 2 S | E | M 1 8 23 M 32	Add MOR register	Backpanel Wiring	Existing ③ +67 add'l parts	40 New Locations	140	65	Impacts speed of other instructions

① Changes required are for adding 10 new parts for the exponent arithmetic and reconfiguring the three boards. However, the EAU would still consist of one control board and two data boards.

② Reconfigure EAU functional schematic but still need only 1 control board and 2 data boards. Forty new memory controller μ code locations needed to handle the extra mantissa word. Thirty-four new parts added for exponent arithmetic and mantissa arithmetic.

③ Reconfigure EAU functional schematic and add hardware to accommodate additional mantissa length. For this format, the EAU will be made up of one control board and three data boards.

77-0819-TA 11

b. ADD TIME (Table 5 only): A comparison of a single precision add time for the addressing mode versus the comparable time for a double word instruction. This number is expressed as a ratio with the double word instruction time being the denominator.

c. Time (Table 4 only): The execution time (inμsec.) required for the instruction.

d. CPU - μ-p: The number of CPU μ-program words required to implement the addressing mode or instruction.

e. MC - μ-p: The number of Memory Controller μ-program words required to implement the addressing mode or instruction.

f. Hardware Required: The additional hardware necessary to implement the addressing mode or instruction on the existing DAIS machine.

g. Physical Changes: The modules in the existing DAIS machine which must be modified to accommodate logic changes in order to implement the addressing mode or instruction.

3.1.2 Changes for Floating-Point Instruction Formats

The changes to the present DAIS computer for the three Floating-Point instruction formats are shown in table 6.

1. Changes required are for adding 10 new parts for the exponent arithmetic and reconfiguring the three boards. However, the EAU would still consist of one control board and two data boards.

2. Reconfigure EAU functional schematic but still need only 1 control board and 2 data boards. Forty new memory controller μ-code locations needed to handle the extra mantissa word. Thirty-four new parts added for exponent arithmetic and mantissa arithmetic.

3. Reconfigure EAU functional schematic and add hardware to accommodate additional mantissa length. For this format the EAU will be made up of one control board and three data boards.

3.2 HARDWARE/FIRMWARE COST SUMMARY

The comparative costs associated with the evaluation results shown in tables 4, 5 and 6 are presented in table 7. Material costs are expressed in 1977 dollars for modifying one computer. Non-recurring costs are expressed in labor hours and include:

a. electrical and micro-code design

b. design verification

c. design documentation

d. printed wiring board artwork changes

Recurring costs are similarly expressed in labor hours and include:

a. assembly and test

b. matrix plate wiring changes

c. system functional verification

d. system acceptance test

3.3 DETAILED DOCUMENTATION

The 20 new instructions for the DAIS machine are listed in table 8. This table also details which micro-code routines are required in the CPU, MC, and EAU. The instruction description, flow charts and timing diagrams for each of the 20 instructions follow table 8.

TABLE 7

COST SUMMARY

COSTS ASSOCIATED WITH TABLE 4

		Parts Cost ($)	Non-Recurring Labor (HR)	Recurring Labor (HR)
1	PSH	4950	1184	109
2	POP	4950	1184	109
3	LPSW	1300	473	55

COSTS ASSOCIATED WITH TABLE 5

		Parts Cost ($)	Non-Recurring Labor (HR)	Recurring Labor (HR)
1	Reg Indr	665	205	16
2	Reg Indr w/Auto Inc	665	50	16
3	Immed Short	1300	430	40
4	Immed Short	710 710	800 750	206 206
5	Jmp Cond IC Rel	710 710	800 750	206 206
6	Jmp Sub IC Rel	710 710	800 750	206 206
7	IC Rel Short	710	790	206
8	6-Bit Base Rel	710 710	850 760	206 206
9	Base Rel Short	710	1000	206
10	Immed Long	710	870	206

77-0819-TA-12

TABLE 8

DETAILED DOCUMENTATION

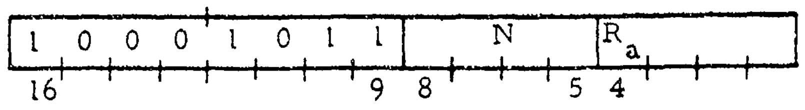
Instruction Name	Instruction Mnemonic	OP Code	Memory Controller - Code Routine	CPU - Code Routine	EAU - Code Routine	Figure Numbers
(1) Push Onto Stack	PSH	88	PSH	PSH	-	5, 6
(2) Pop From Stack	POP	98	POP	POP		7, 8
(3) Lead Program Status Words	LPSW	8E	LPSW	CLPSW		9, 10, 11
(4) Floating Point Add, Register to Register	FAR	A8	TPRS	FA	FA	12, 13, 14, 15
(5) Floating Subtract, Register to Register	FSR	B8	TPRS	FS	FS	16, 17, 18, 19
(6) Floating Multiply, Register to Register	FMR	C8	TPRS	FM	FM	20, 21, 22, 23
(7) Floating Divide, Register to Register	FDR	D8	TPRS	FD	FD	24, 25, 26, 27
(8) Floating Compare, Register to Register	FCR	F8	TPRS	FC	-	28, 29, 30
(9) Single Precision Multiply	M	CO	TPDEX	M	M	31, 32, 33, 34
(10) Single Precision Multiply, Register to Register	MR	C1	TPR	MR	MR	35, 36, 37, 38
(11) Single Precision Multiply, Indirect	MI	C2	TPIEX	M	M	39, 40, 41, 42
(12) Single Precision Divide	D	D0	TPDEX	D	D	43, 44, 45, 46
(13) Single Precision Divide, Register to Register	DR	D1	TPR	DR	DR	47, 48, 49, 50
(14) Single Precision Divide, Indirect	DI	D2	TPIEX	D	D	51, 52, 53, 54
(15) Double Precision Absolute Values Register to Register	DABS	AC	TPR	DABS	-	55, 56, 57
(16) Negate Double Precision Register	DNEG	BC	TPR	DNEG	-	58, 59, 60
(17) Shift Right Cyclic	SRC	64	TPR	SRX	-	61, 62, 63
(18) Double Shift Left Logical	DSLL	65	TPR	DSLX	-	64, 65, 66
(19) Double Shift Right Arithmetic	DSRA	67	TPR	DSRX	-	67, 68, 69
(20) Double Shift Right Cylic	DSRC	69	TPR	DSRX	-	70, 71, 72

77-0819 TA 13

MNEMONIC: PSH **OP CODE:** 8B

SHORT NAME: push onto stack

FORMAT: PSH N, R_A



DESCRIPTION: The contents of registers R_a through $R_{(a+N)}$ are pushed onto a stack in memory using R15 as the stack pointer. When completed, R15 is incremented by N+1.

If N=0, then <u>only</u> R_a is pushed onto the stack.

REGISTERS AFFECTED: R15

TIMING: (3.0 + 1.6N) μsec

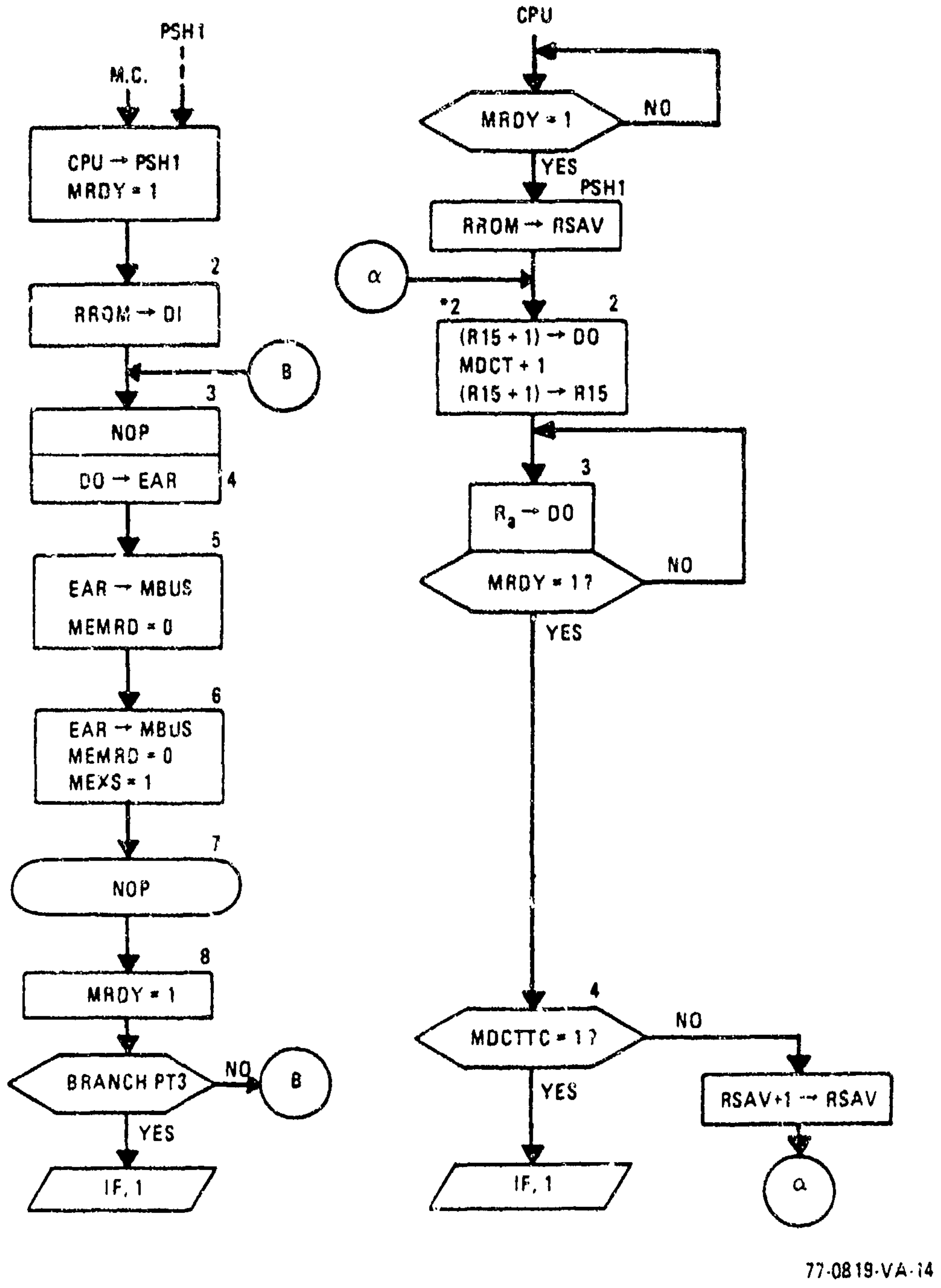


Figure 5, PSH Instruction

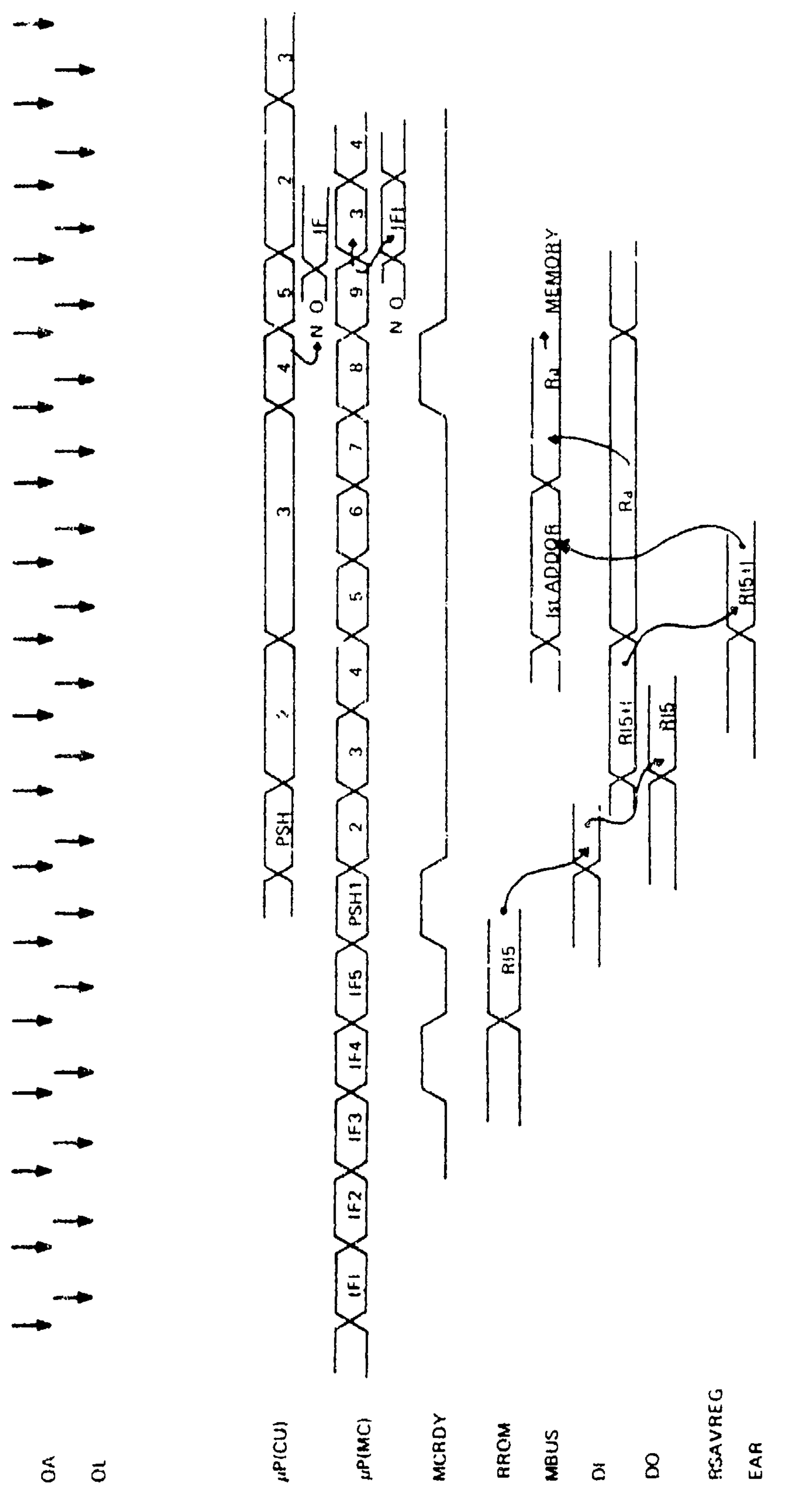


Figure 6 . PSII Timing Diagram

MNEMONIC: POP OP CODE: 9B

SHORT NAME: pop from stack

FORMAT: POP N, R_A

1	0	0	1	1	0	1	1	N	R_a
16							9	8 5	4 1

DESCRIPTION: Register R_a through $R_{(a-N)}$ are loaded sequentially from the stack in memory using R15 as the stack pointer. When completed, R15 is loaded with (R15-N-1). The CS register is set for each word transferred. If N=0, then only R_a will be loaded.

REGISTERS AFFECTED: R_a through $R_{(a-N)}$, R15, CS

TIMING: (3.0 + 1.6N) μsec

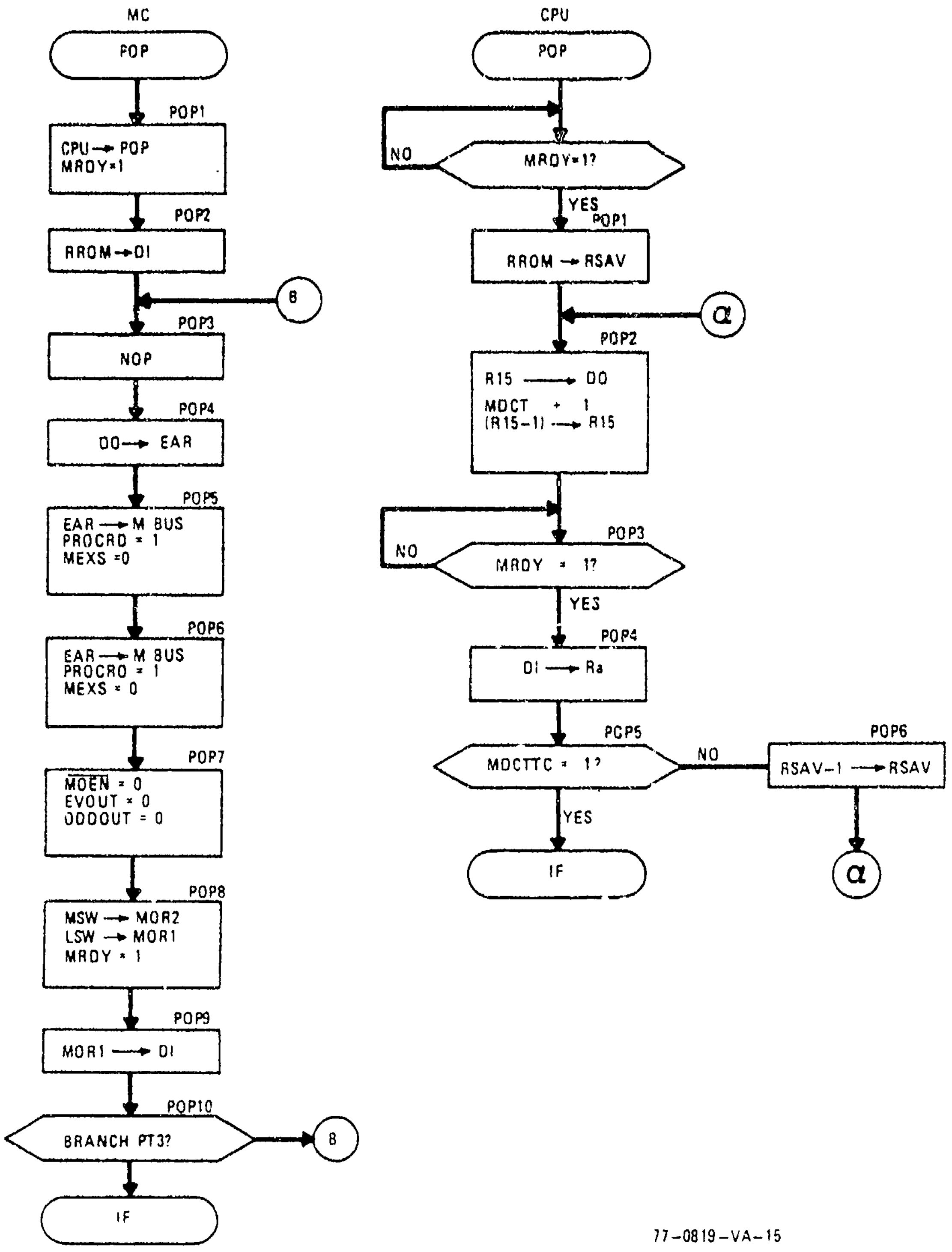


Figure 7, POP Instruction

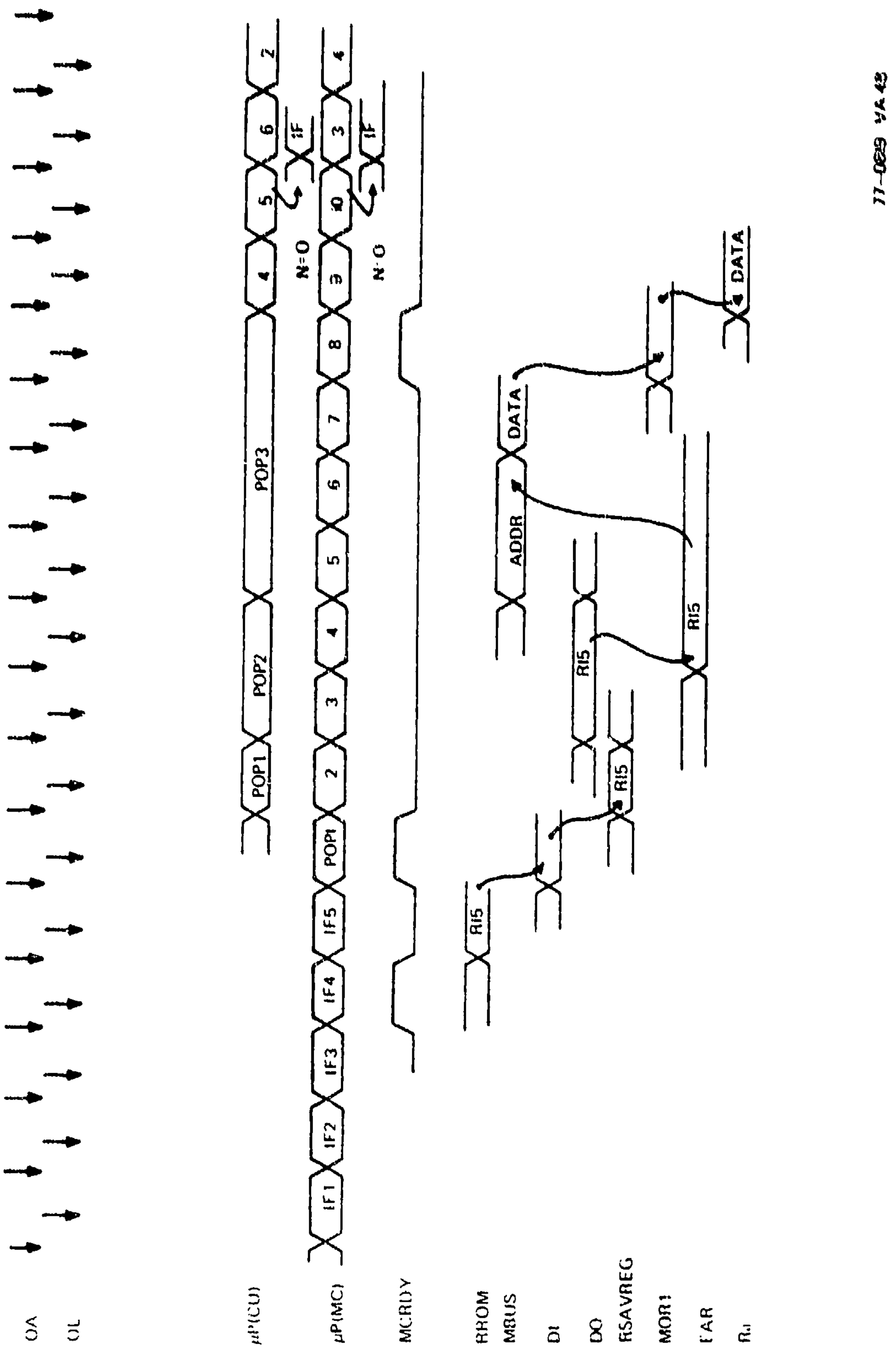
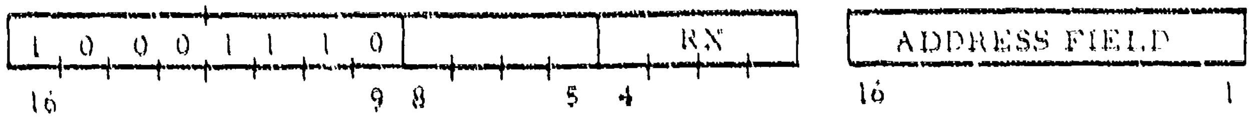


Figure 8. POP Timing Diagram

MNEMONIC: LPSW OP CODE: 8E

SHORT NAME: load program status words

FORMAT:	LPSW	ADDR	non-indexed
	LPSW	ADDR, RX	indexed



DESCRIPTION: The current three program status words are replaced by three sequential memory words located at the effective address.

This instruction is used for context switching and as a return from interrupt.

REGISTERS AFFECTED: IC, CS

TIMING: 4.4 μsec

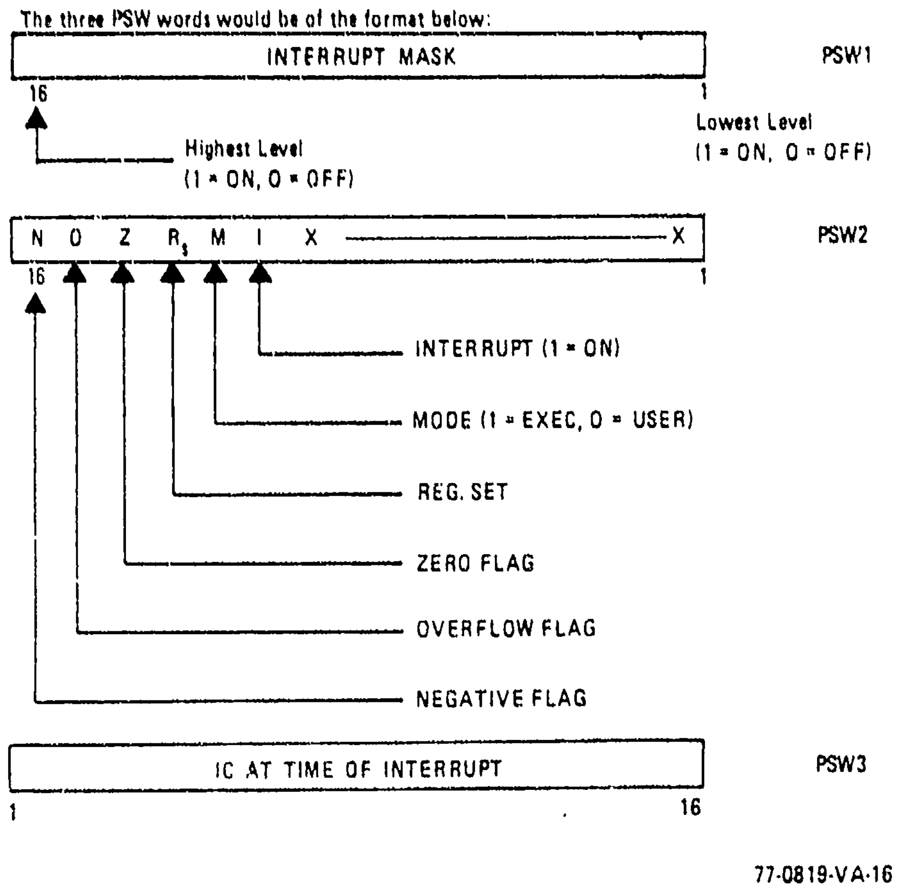


Figure 9. LPSW Words

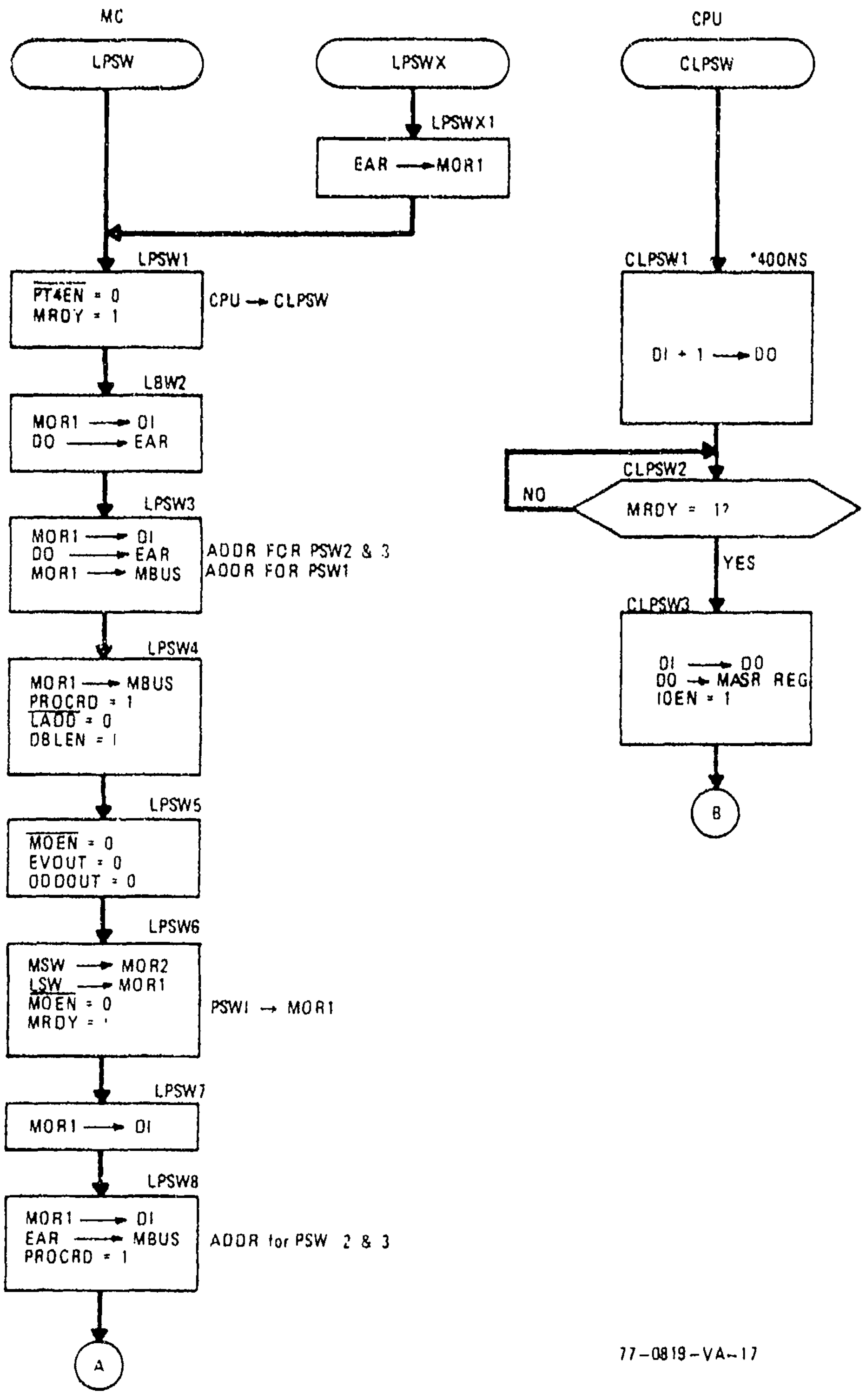


Figure 10. LPSW Instruction

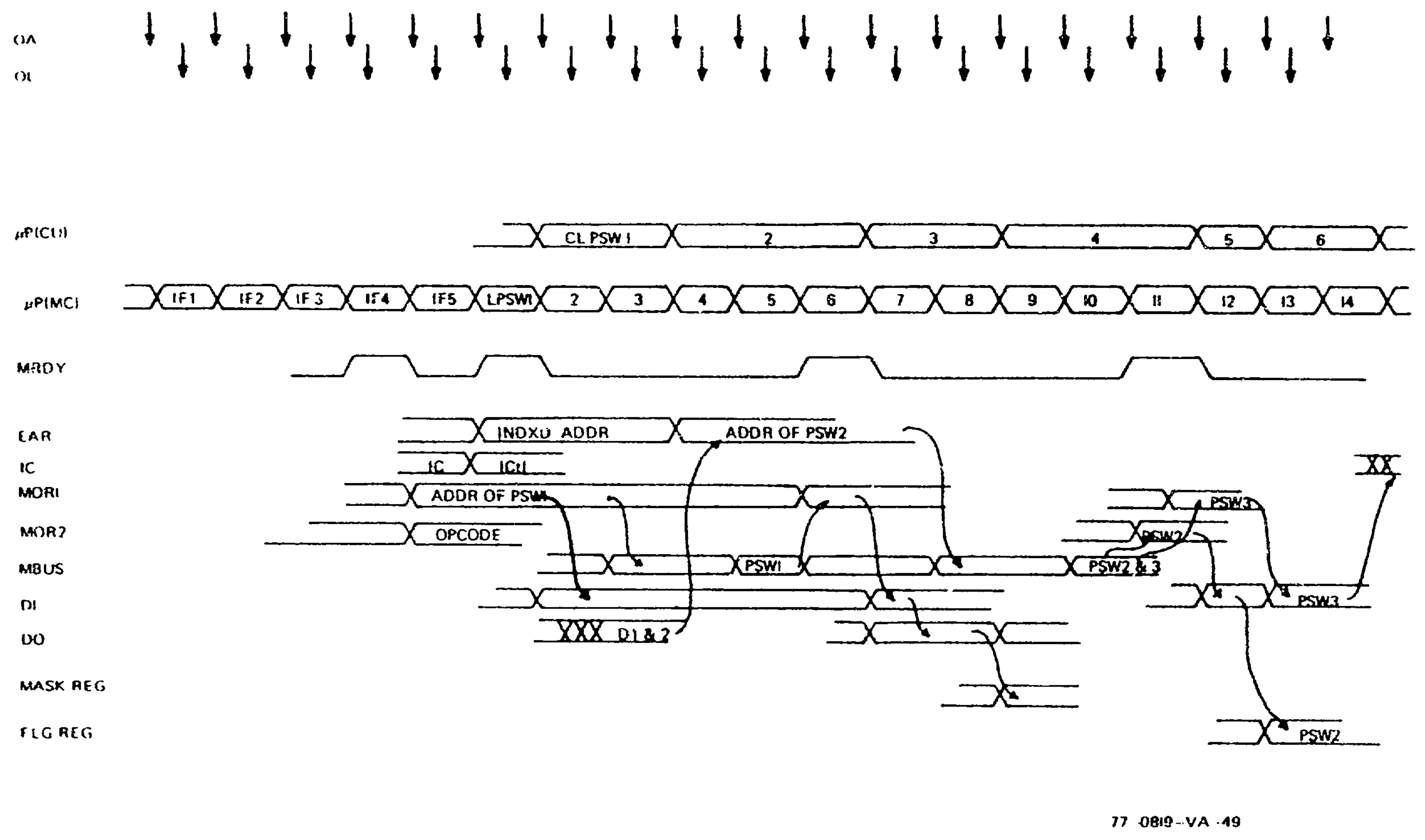


Figure 11. LPSW Timing Diagram

MNEMONIC: FAR

OP CODE: A8

SHORT NAME: floating point ADD, register-to-register

FORMAT: FAR R1, R2

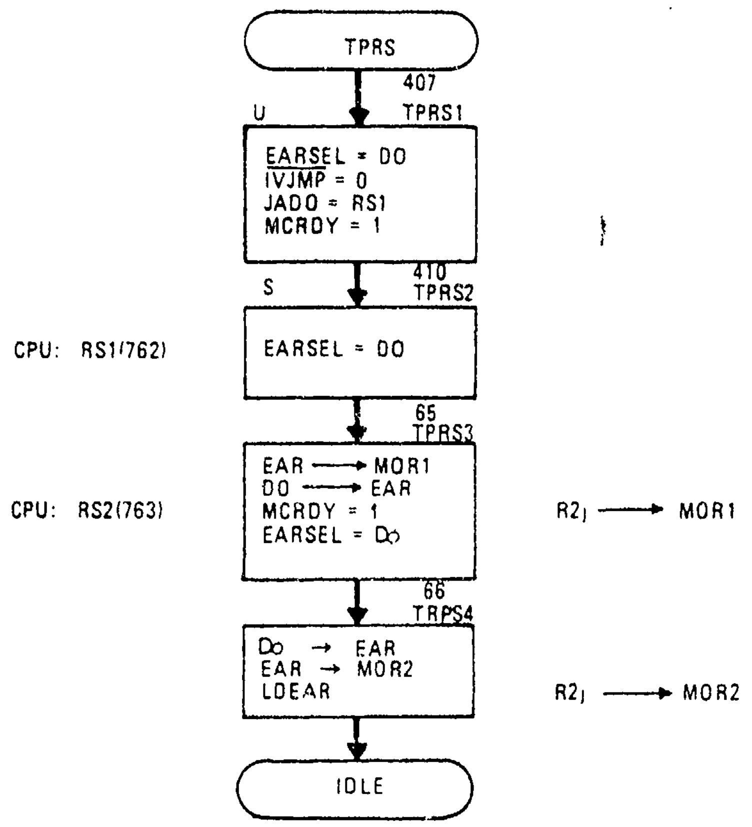
1	0	1	0	1	0	0	0	R1	R2

DESCRIPTION: The floating point number in registers R2 and R2 plus one is a dded to the content of registers R1 and R1 plus one. The conditions status, CS, is set based on the floating point result in registers R1 and R1 + 1 and overflow. Overflow is defined as exponent overflow or underflow during the operation. Upon overflow or underflow a floating point zero, 00000080, is the result, R1 and R2 must be even.

REGISTERS AFFECTED: R1, R1 1, CS

TIMING: 4.2

TYPE – PS (REGISTER TO REGISTER SPECIAL)



77-0819-VA-18

Figure 12 , TYPE - PS (Register to Register Special)

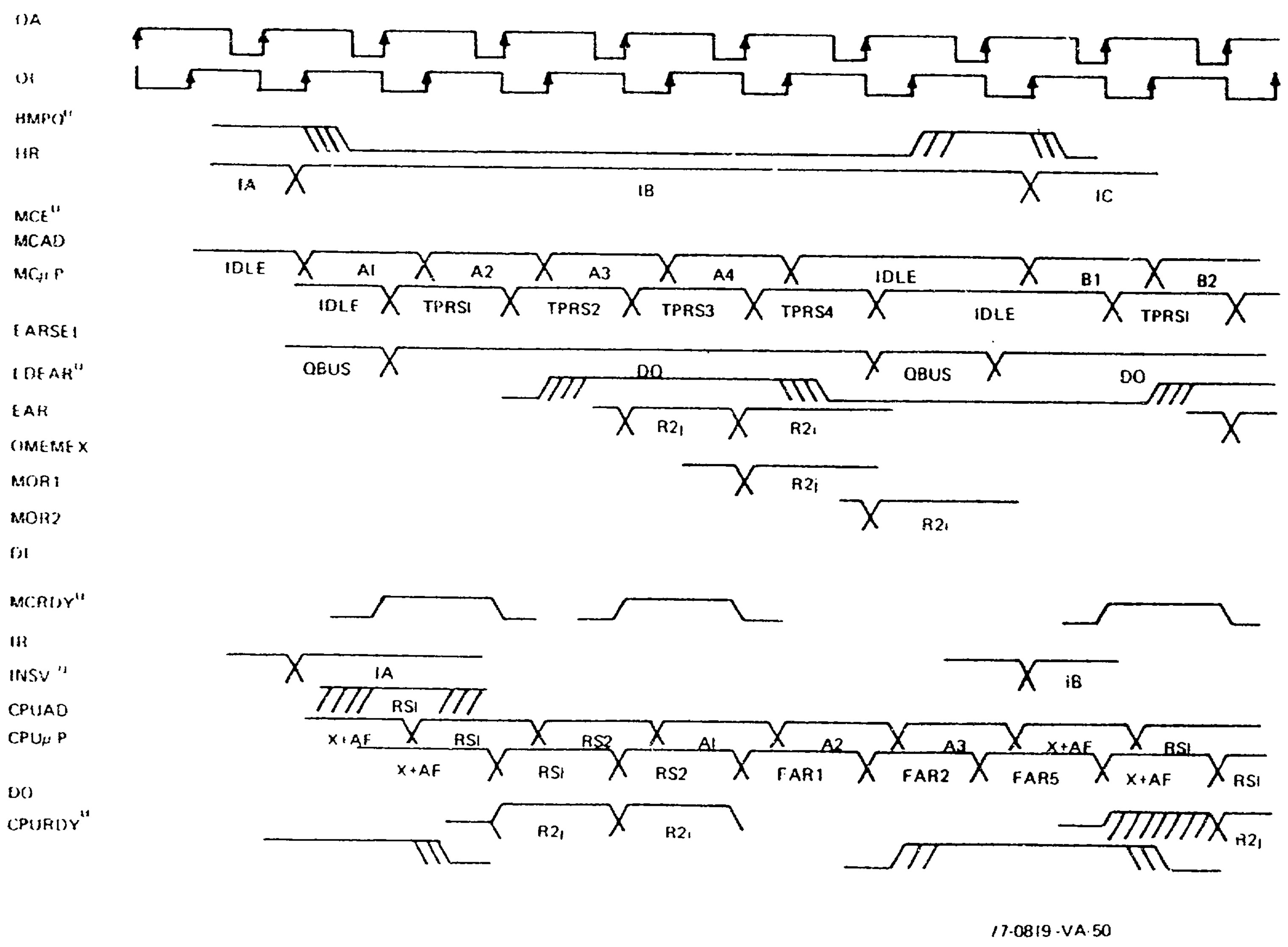
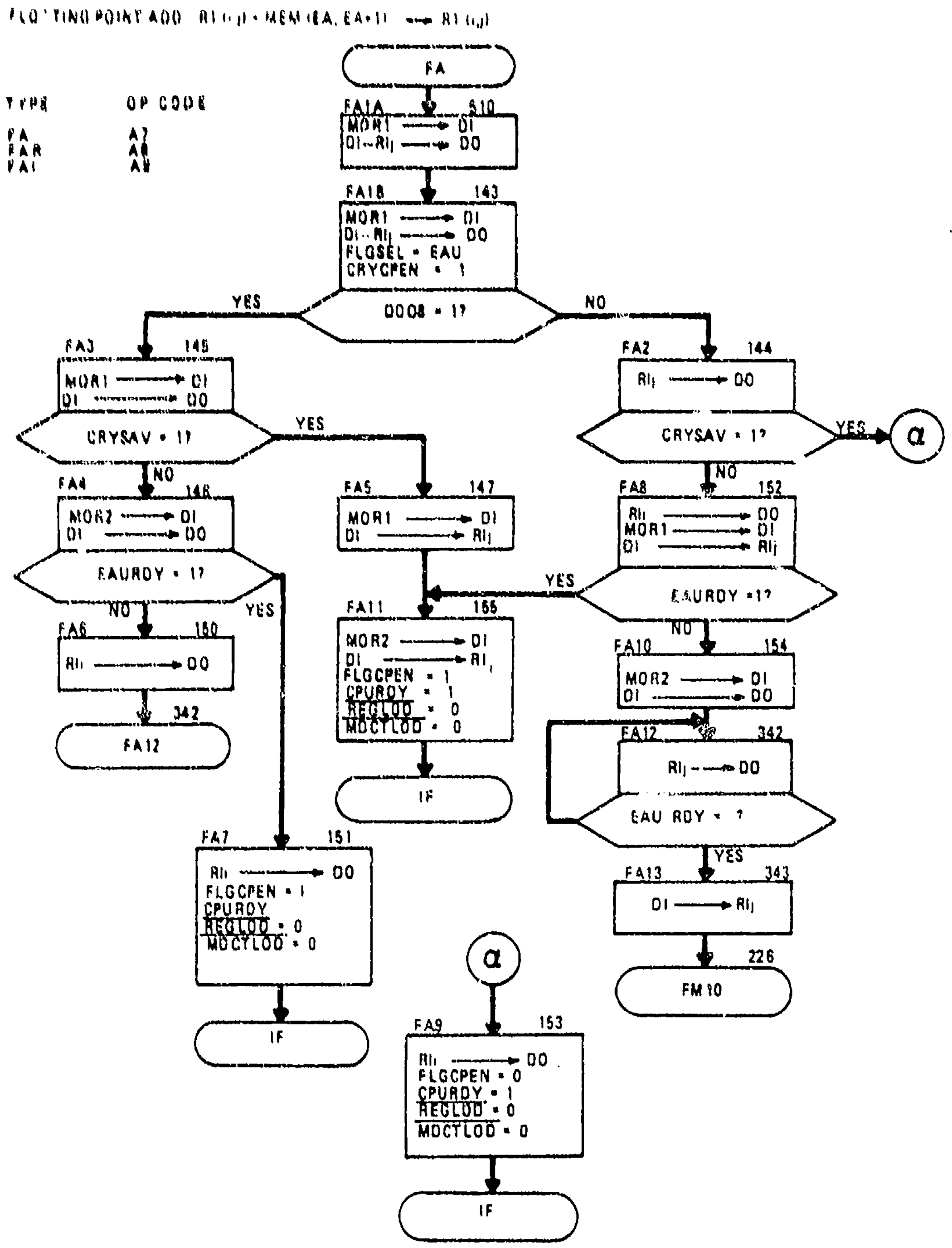


Figure 13. FAR Timing Diagram



77-0819-VA-19

Figure 14 . FAR Instruction

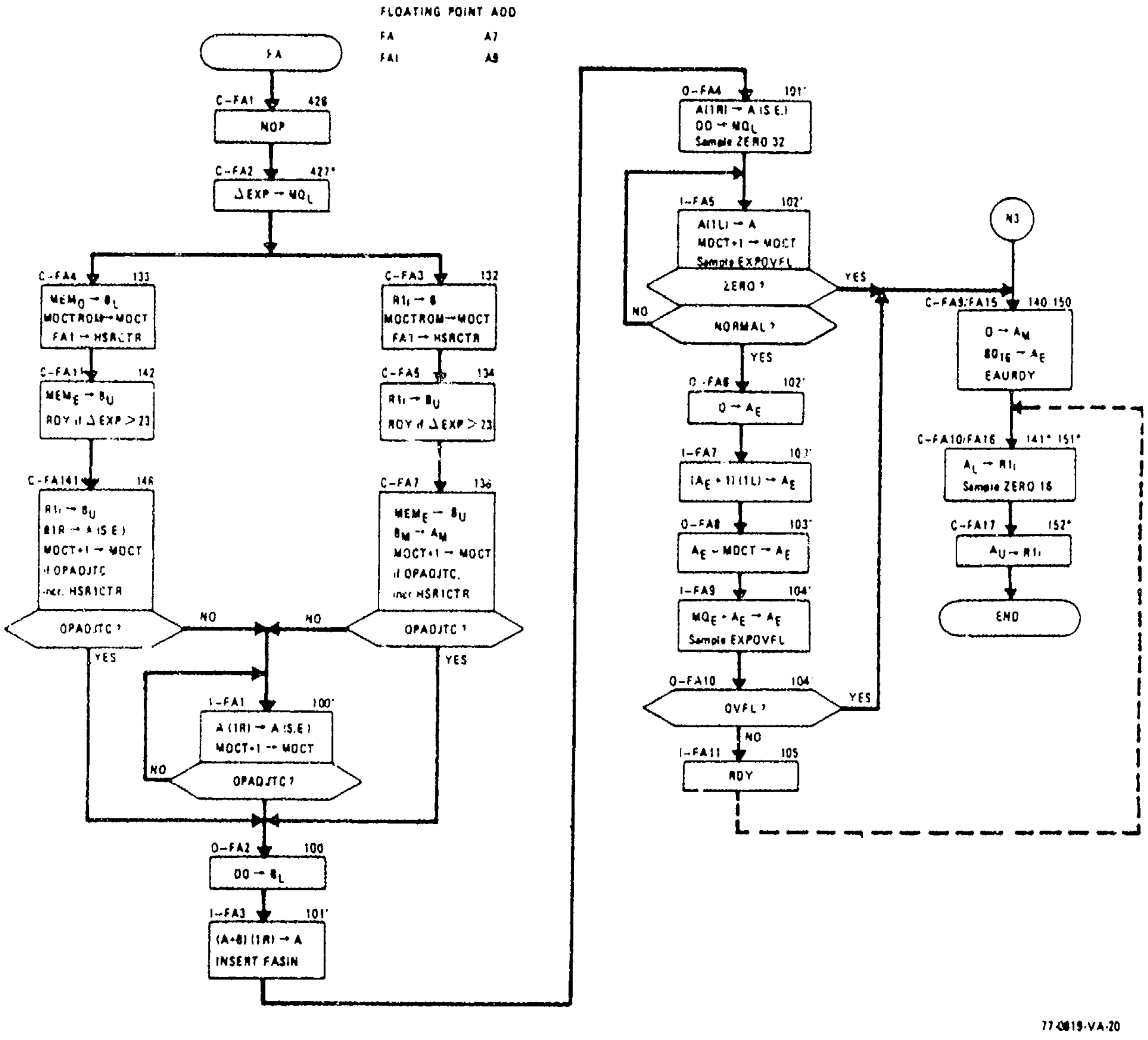


Figure 15 . FAR Instruction

MNEMONIC: FSR **OP CODE:** B8

SHORT NAME: floating subtract, register-to-register

FORMAT: FSR R1, R2

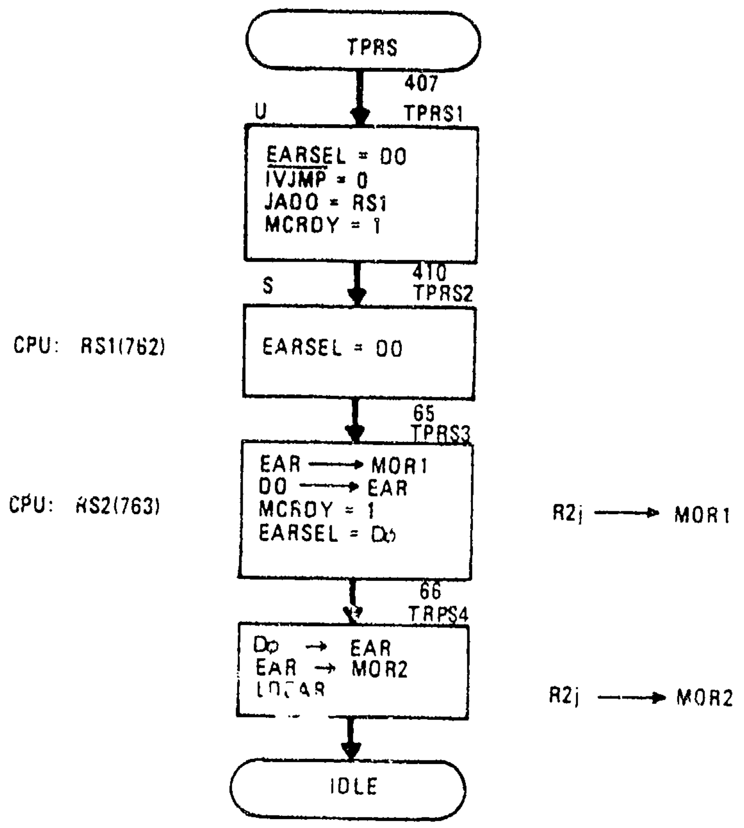
1	0	1	1	1	0	0	0	R1	R2

DESCRIPTION: The floating point number in register R2 and R2 + 1 is subtracted from the floating point number in register R1 and register R1+ 1. The difference remains in registers R1 and R1+ 1. The condition status, CS, is set based on the floating point result in registers R1 and R1+ 1 and overflow. Overflow is defined as exponent overflow or underflow during the operation. Upon overflow or underflow a floating point zero, 00000080, is the result. R1 and R2 must be even.

REGISTERS AFFECTED: R1, R1 + 1, CS

TIMING: 4. 2

TYPE – PS (REGISTER TO REGISTER SPECIAL)



77–0819–VA–18

Figure 16 . TYPE - PS (Register to Register Special)

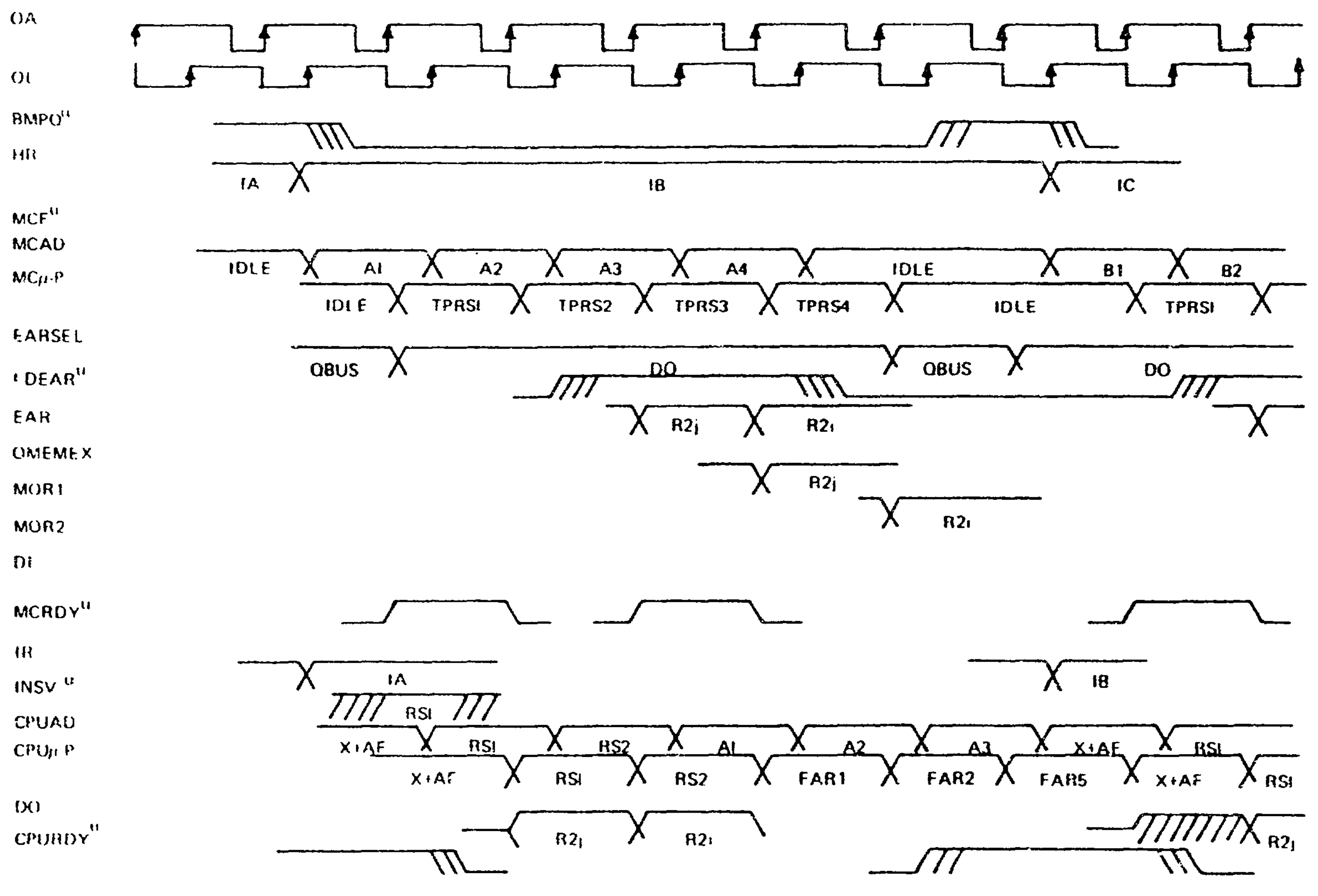
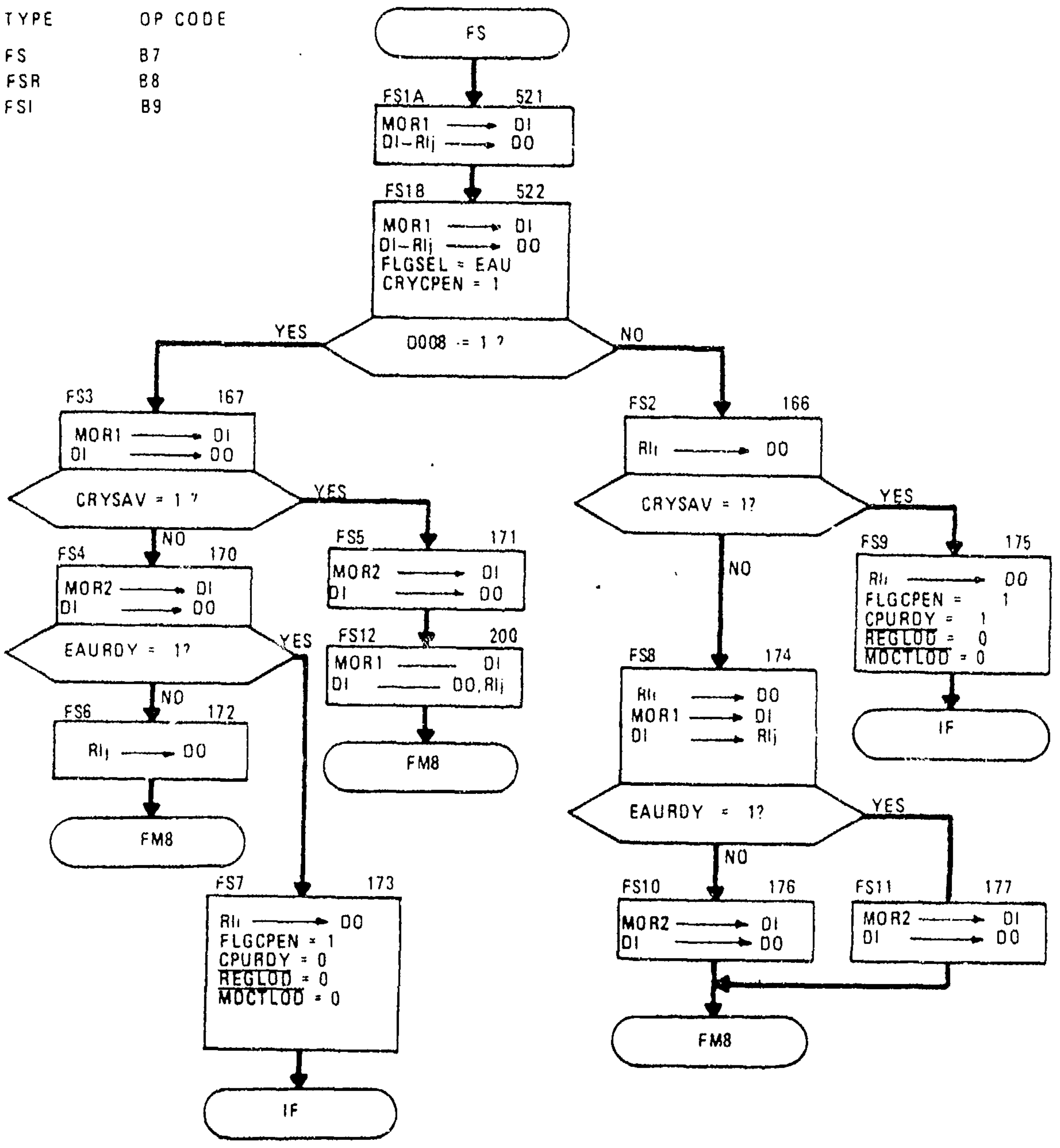


Figure 17 . FSR Timing Diagram

FLOATING POINT SUBTRACT, R1 (i,j) – MEM (EA, EA+1) → R1 (i,j)

TYPE	OP CODE
FS	B7
FSR	B8
FSI	B9



77-0819-VA-21

Figure 18 . FSR Instruction

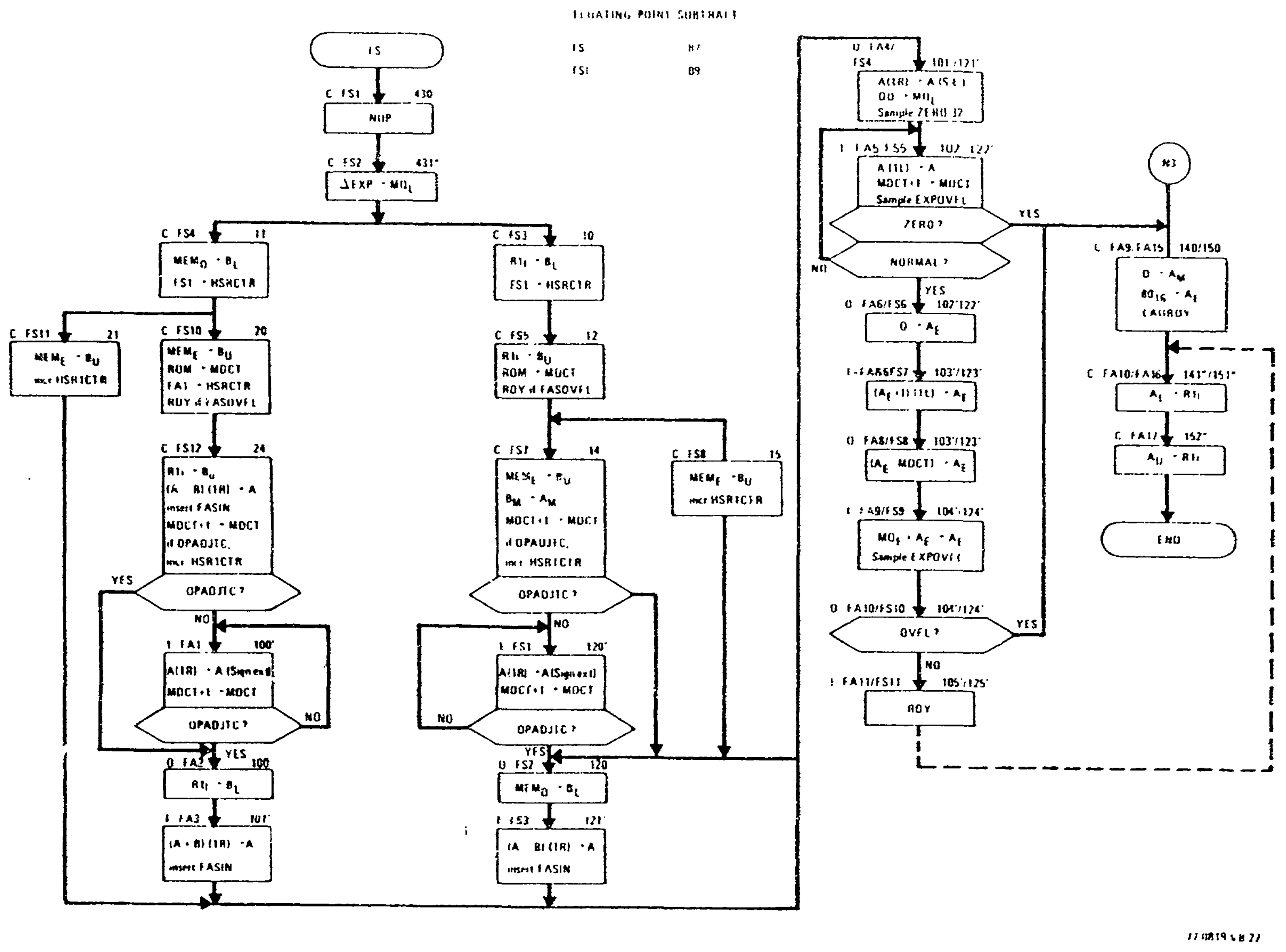

FLOATING POINT SUBTRACT
FS 87
FS1 89
FS
C FS1 430 NOP
C FS2 431' ΔEXP · MQL
C FS4 11 MEMQ · BL FS1 · HSRCTR
C FS3 10 RIL · BL FS1 · HSRCTR
C FS11 21 MEME · BU incr HSRICTR
C FS10 20 MEME · BU ROM · MOCT FA1 · HSRCTR RDY if FASOVFL
C FS5 12 RIL · BU ROM · MOCT RDY if FASOVFL
C FS12 24 RIL · BU (A BI(IR) · A insert FASIN MOCT+1 · MOCT if OPADJTC, incr HSRICTR
C FS7 14 MEME · BU BM · AM MOCT+1 · MOCT if OPADJTC, incr HSRICTR
C FS8 15 MEME · BU incr HSRICTR
OPADJTC?
YES
NO
1 FA1 100' A(IR) · A (Sign ext) MOCT+1 · MOCT
1 FS1 120' A(IR) · A(Sign ext) MOCT+1 · MOCT
OPADJTC?
0 FA2 100 RIL · BL
0 FS2 120 MEMQ · BL
1 FA3 101' (A + B)(IR) · A insert FASIN
1 FS3 121' (A BI(IR) · A insert FASIN
0 FA4/FS4 101'/121' A(IR) · A(S E) OD · MQL Sample ZERO 32
1 FA5 FS5 102'/122' A(IL) · A MOCT+1 · MOCT Sample EXPOVFL
ZERO?
NORMAL?
0 FA6/FS6 102'/122' 0 · AE
1 FA6/FS7 103'/123' (AE+1)(IL) · AE
0 FA8/FS8 103'/123' (AE MOCT) · AE
1 FA9/FS9 104'/124' MQE + AE · AE Sample EXPOVFL
0 FA10/FS10 104'/124'
OVFL?
1 FA11/FS11 105'/125' RDY
N3
C FA9:FA15 140/150 0 · AM 80₁₆ · AE CAGRDY
C FA10:FA16 141'/151' AE · RIL
C FA17 152' AU · RIL
END
77 0819 V 8 27

Figure 19. FSR Instruction

MNEMONIC: FMR OP CODE: C8

SHORT NAME: floating multiply, register-to-register

FORMAT: FMR R1, R2

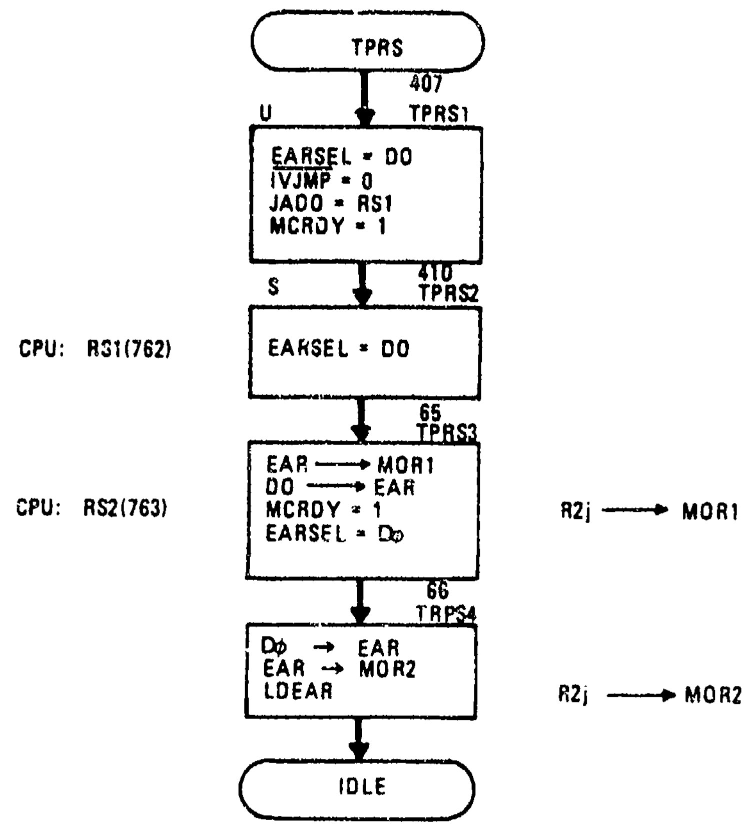
1	1	0	0	1	0	0	0	R1	R2

DESCRIPTION: The floating point number in registers R2 and R2+1 is multiplied by the floating point number in registers R1 and R1 + 1. The floating point result is retained in registers R1 and R1+1. The condition status, CS, is set based on the floating point result in registers R1 and R1+1 and overflow. Overflow is defined as exponent overflow or underflow during the operation. Upon overflow or underflow a floating point zero. 00000080, is the result. R1 and R2 must be even.

REGISTERS AFFECTED: R1, R1+1, CS

TIMING: 5.6

TYPE – PS (REGISTER TO REGISTER SPECIAL)



77-0819-VA-18

Figure 20 . TYPE - PS (Register to Register Special)

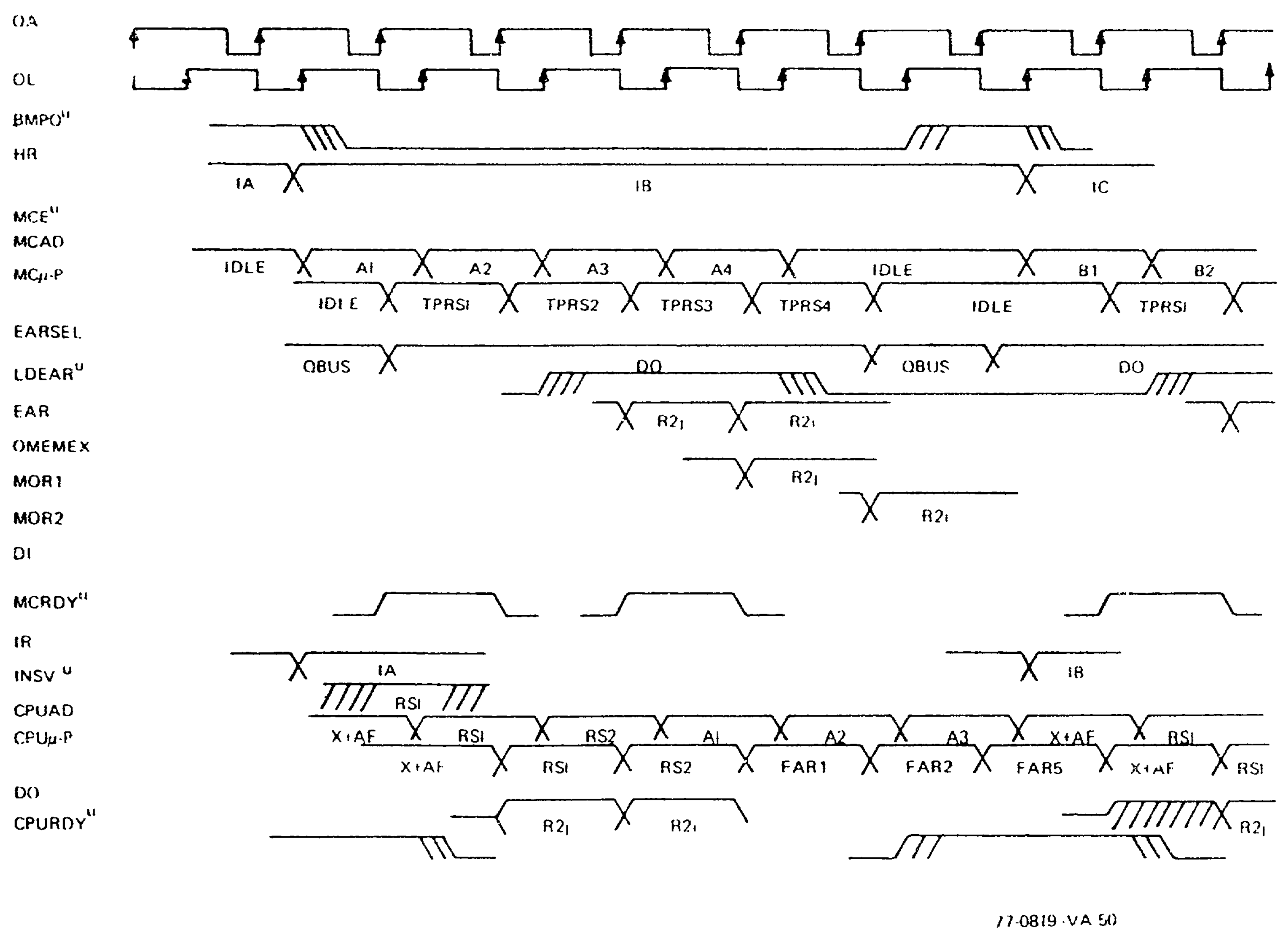
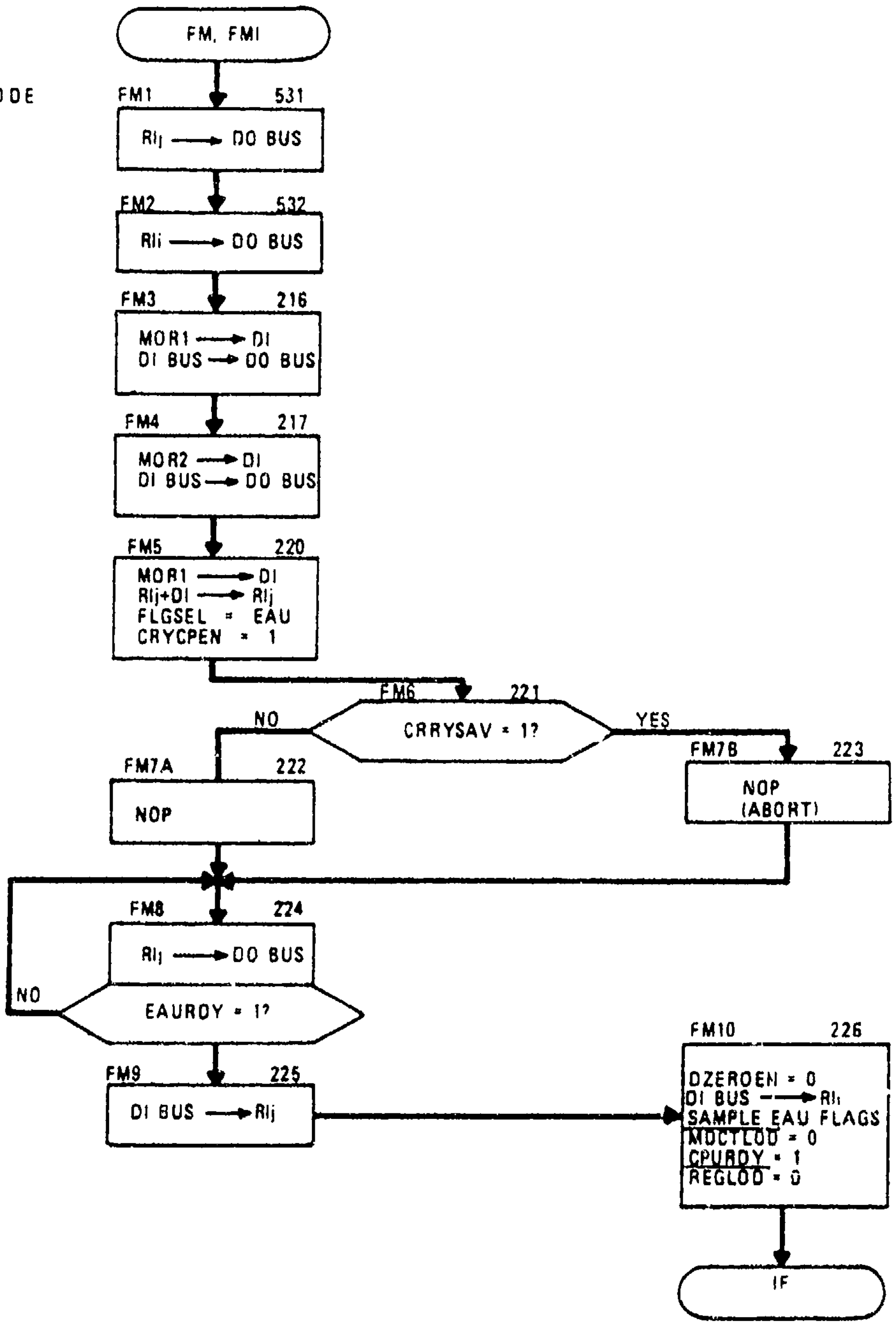


Figure 21'. FMR Timing Diagram

FLOATING POINT MULTIPLY: (R1i, R1j) * (MEMORY, MEMORY + 1) → R1i, R1j

TYPE	OP CODE
FM	C7
FMR	C8
FMI	C9



77-0819-VA-23

Figure 22 . FMR Instruction

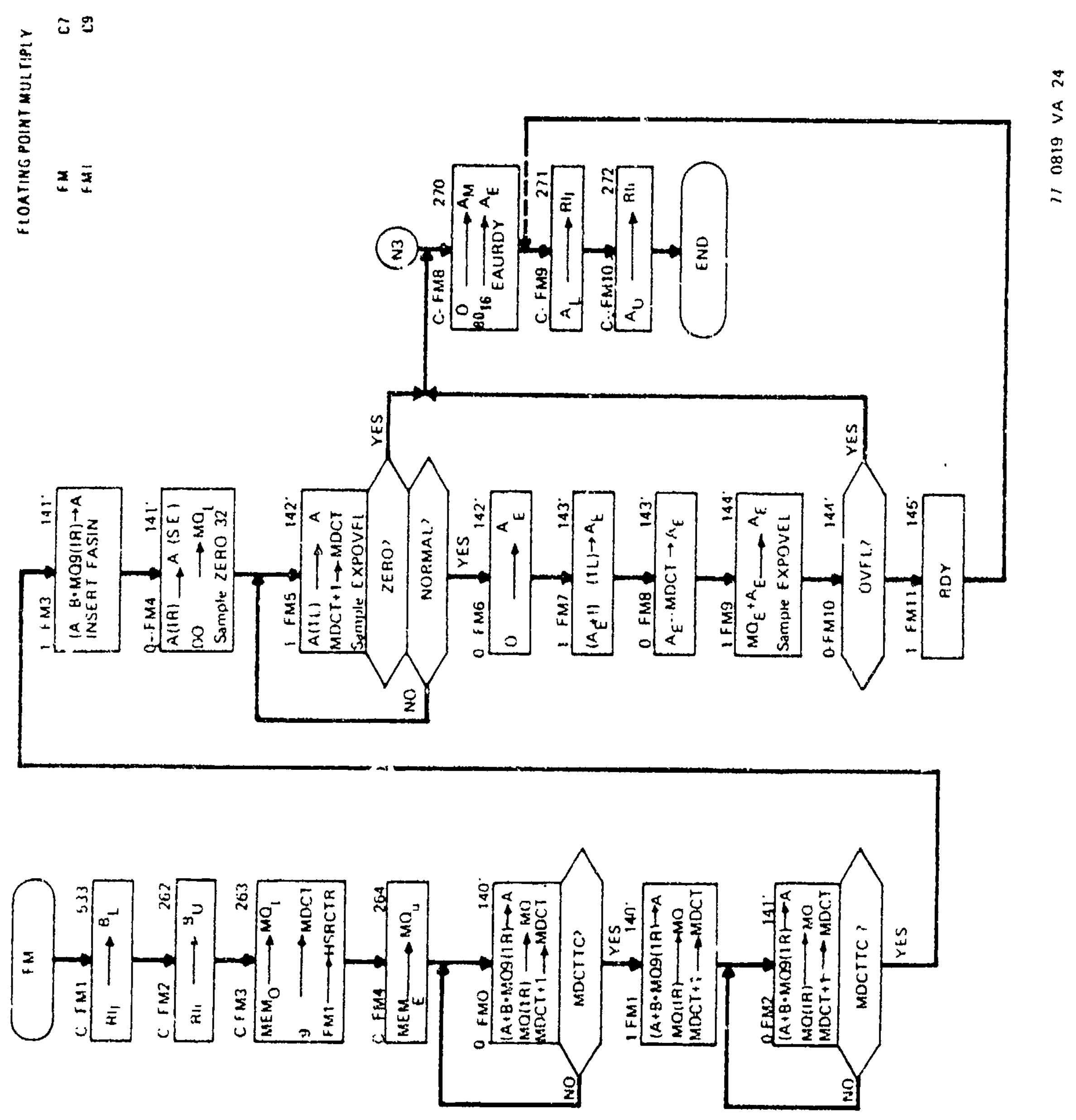


Figure 23. FMR Instruction

MNEMONIC: FDR OP CODE: D8

SHORT NAME: floating divide, register-to-register

FORMAT: FDR R1, R2

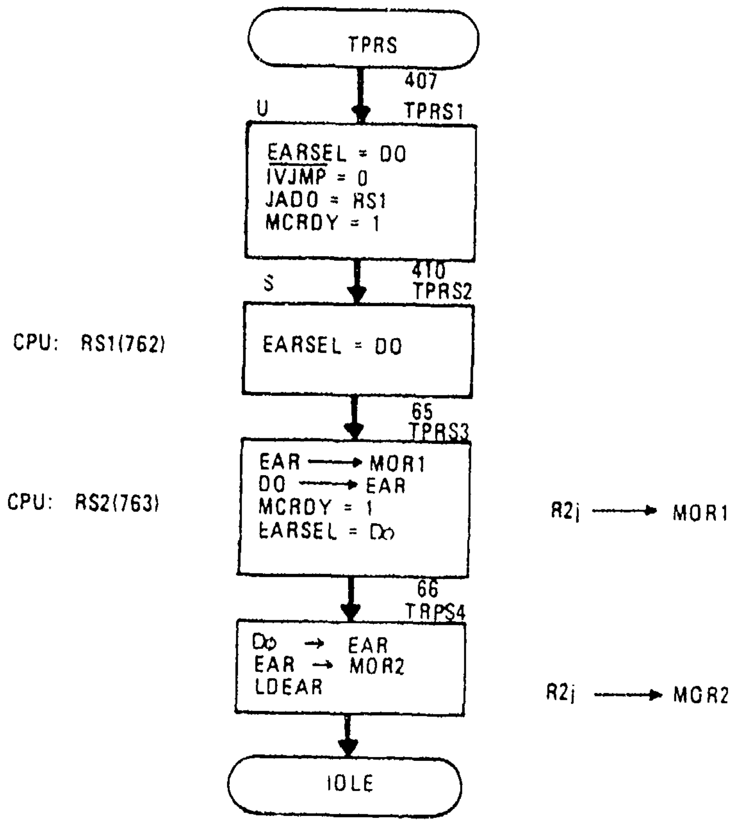
1	1	0	1	1	0	0	0	R1	R2

DESCRIPTION: The floating point number in registers R1 and R1+1 is divided by the floating point number in registers R2 and R2+1. The floating point quotient is retained in registers R1 and R1+1. The condition status, CS, is set based on the floating point result in registers R1 and R1+1 and overflow. Overflow is defined as exponent overflow or underflow during the operation. Upon overflow or underflow a floating point zero, 00000080, is the result. R1 and R2 must be even.

REGISTERS AFFECTED: R1, R1+1, CS

TIMING: 6.0

TYPE – PS (REGISTER TO REGISTER SPECIAL)



77-0819-VA-18

Figure 24 . TYPE - PS (Register to Register Special)

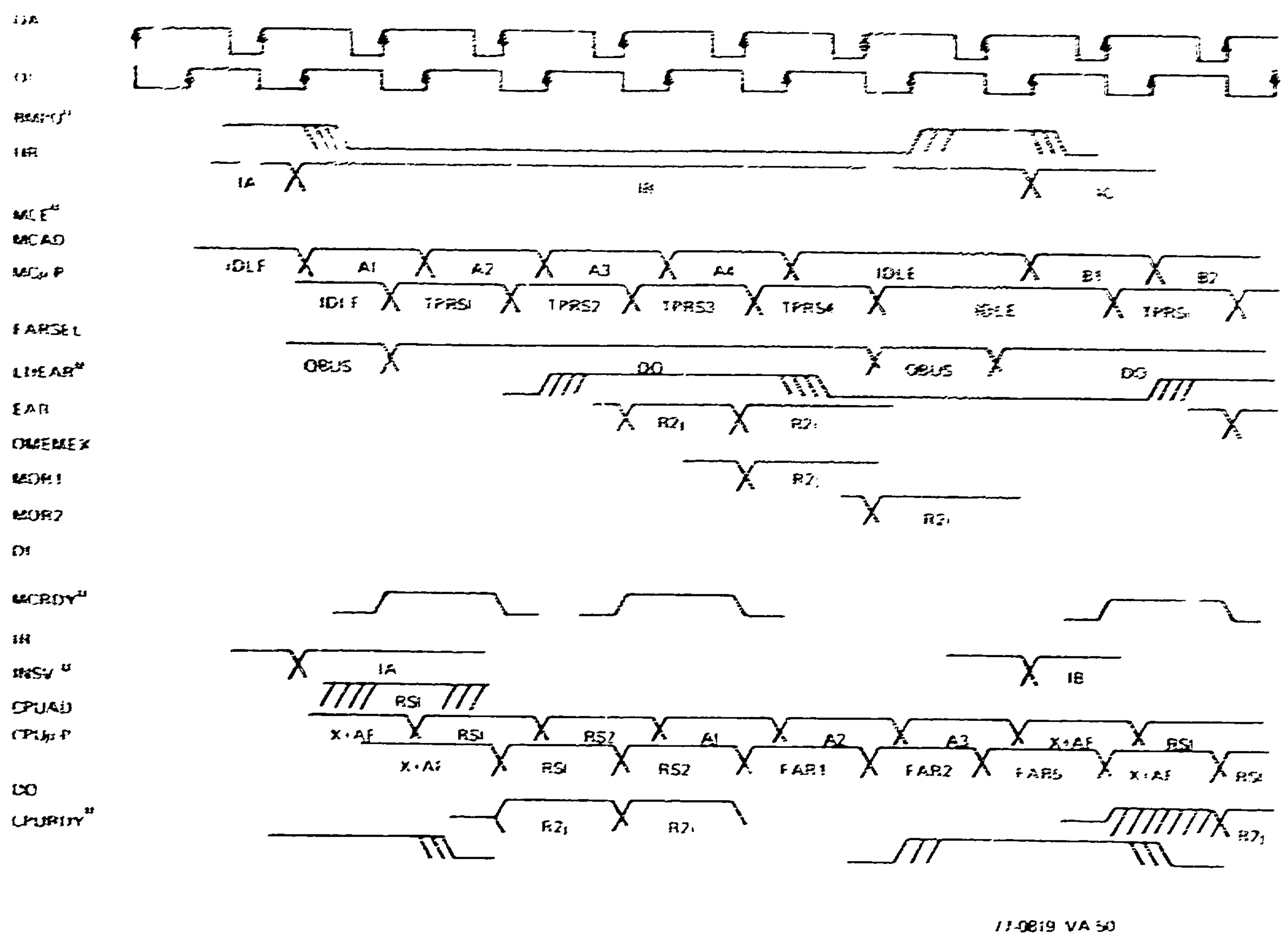


Figure 25. FDR Timing Diagram

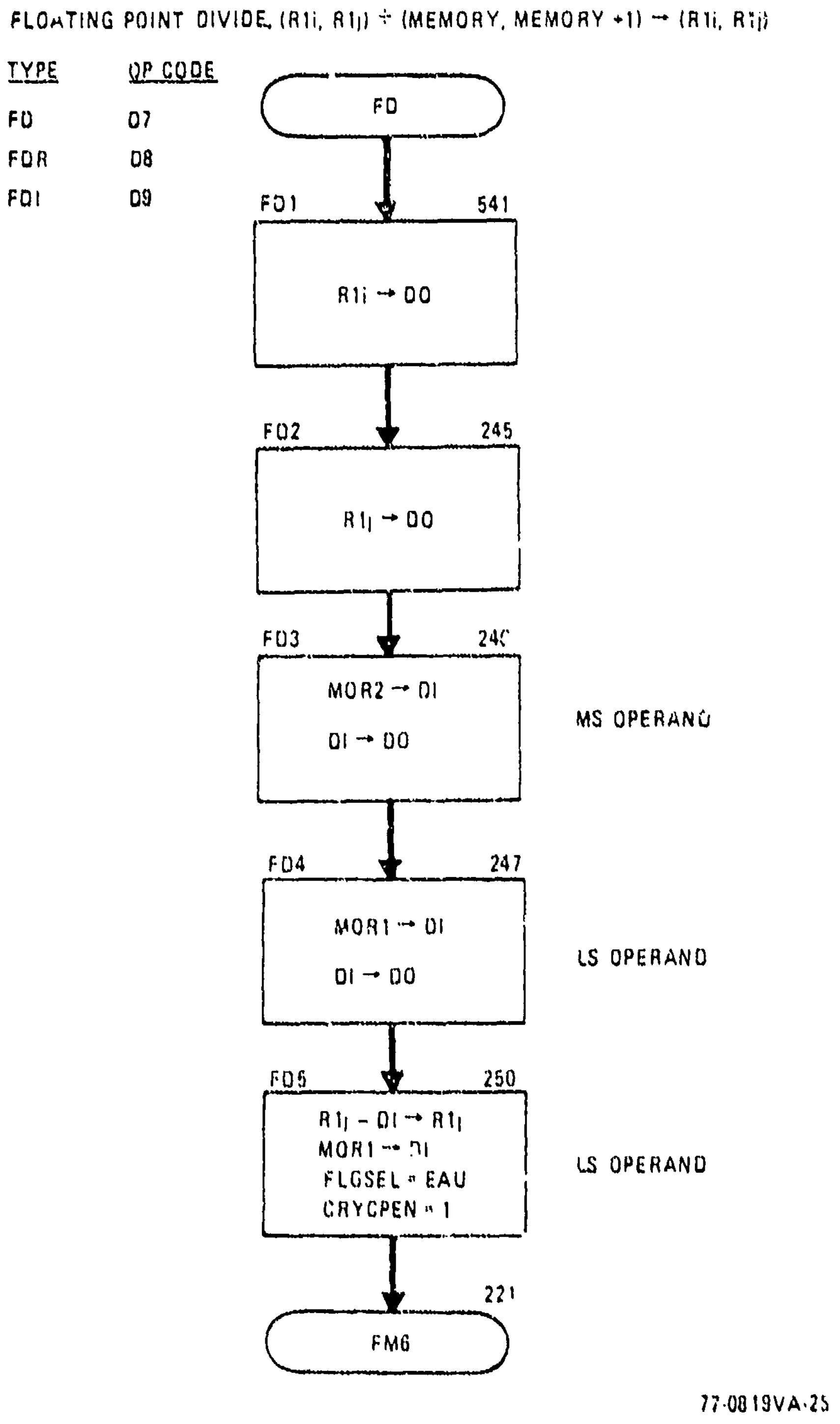


Figure 2… . FDR Instruction

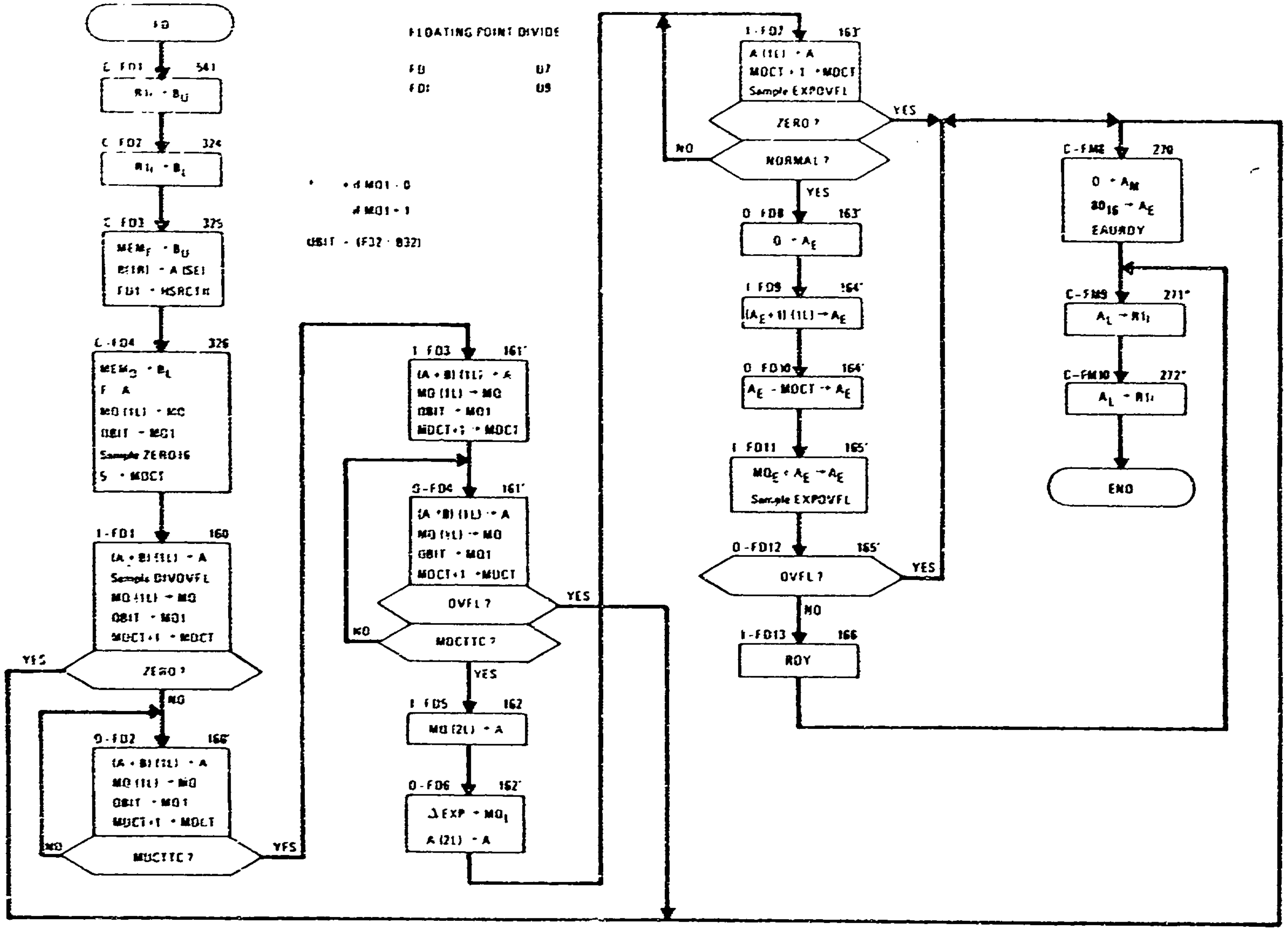


Figure 27. FDR Instruction

MNEMONIC: FCR OP CODE: F8

SHORT NAME: floating compare, register-to-register

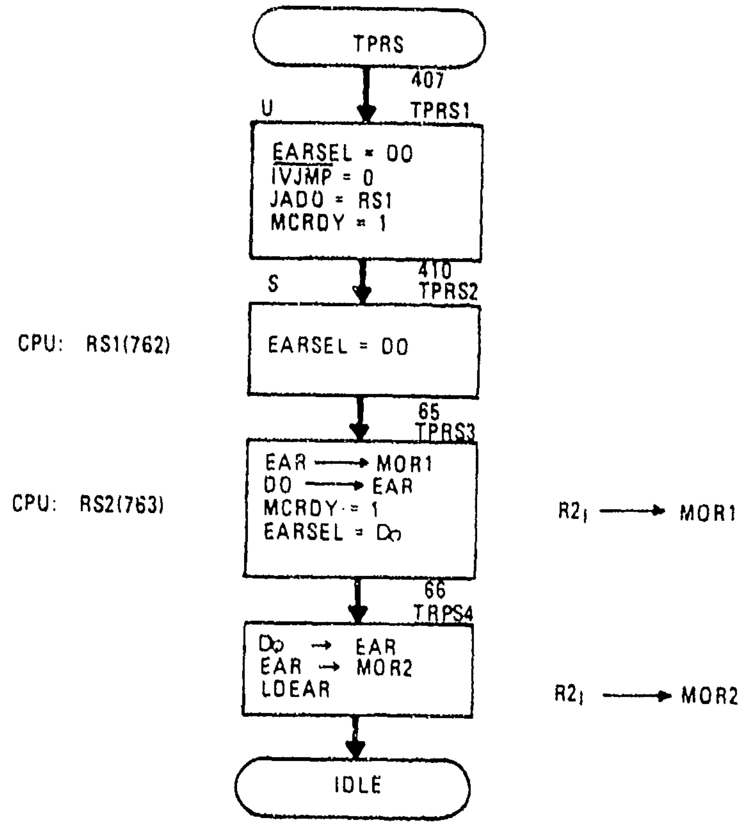
FORMAT: FCR R1, R2

1	1	1	1	1	0	0	0	R1	R2

DESCRIPTION: The floating point number in registers R1 and R1+1 is compared to the floating point number in registers R2 and R2+1. If R1<R2 then the condition status, CS, is set to 1 (less than). If R1 = R2 then CS is set to 2 (equal to). If R1>R2 then CS is set to 4 (greater than). No registers are changed. R1 and R2 must be even.

REGISTERS AFFECTED: CS

TYPE – PS (REGISTER TO REGISTER SPECIAL)



77–0819–VA–18

Figure 28 . TYPE - PS (Register to Register Special)

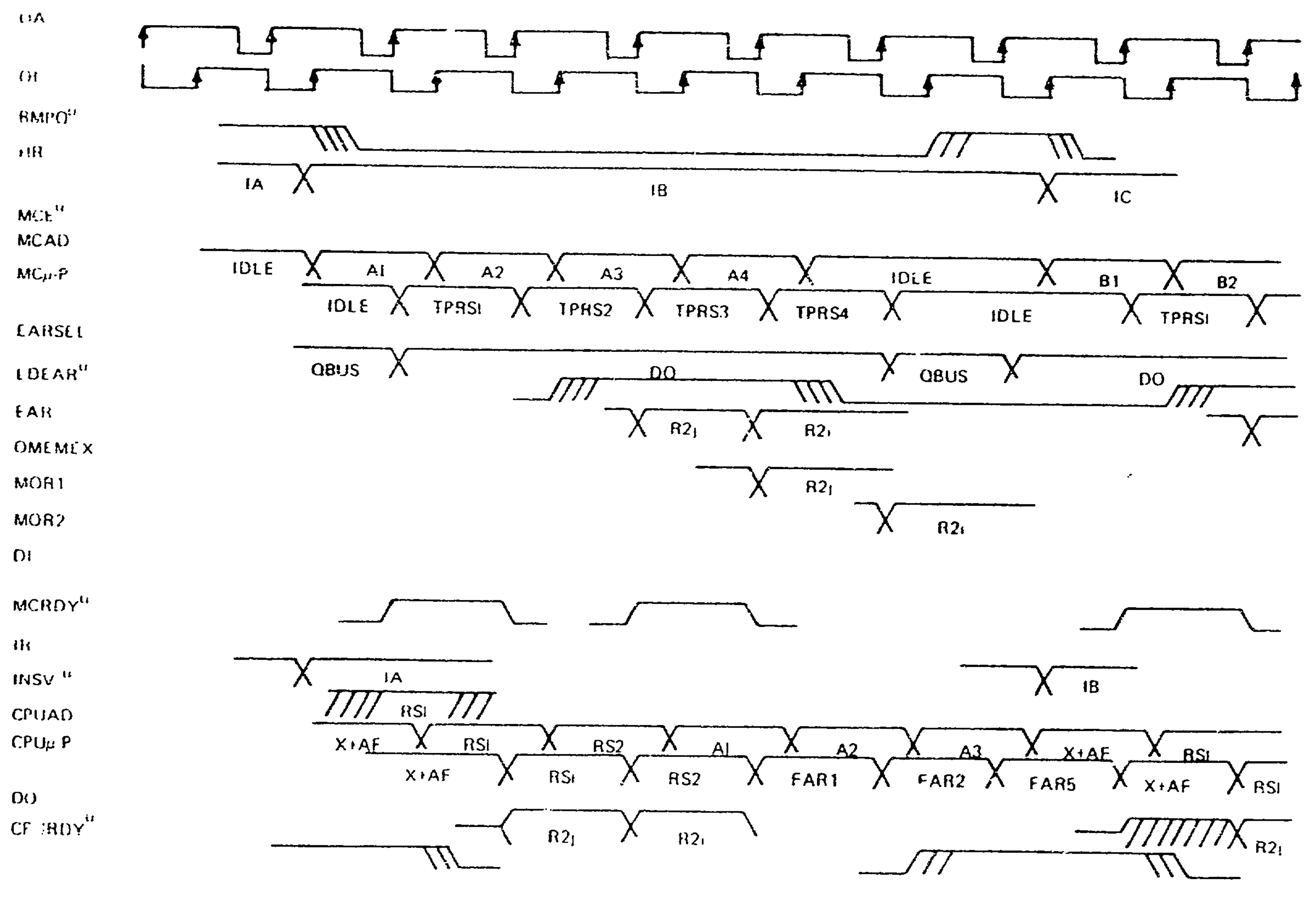


Figure 29. FCR Timing Diagram

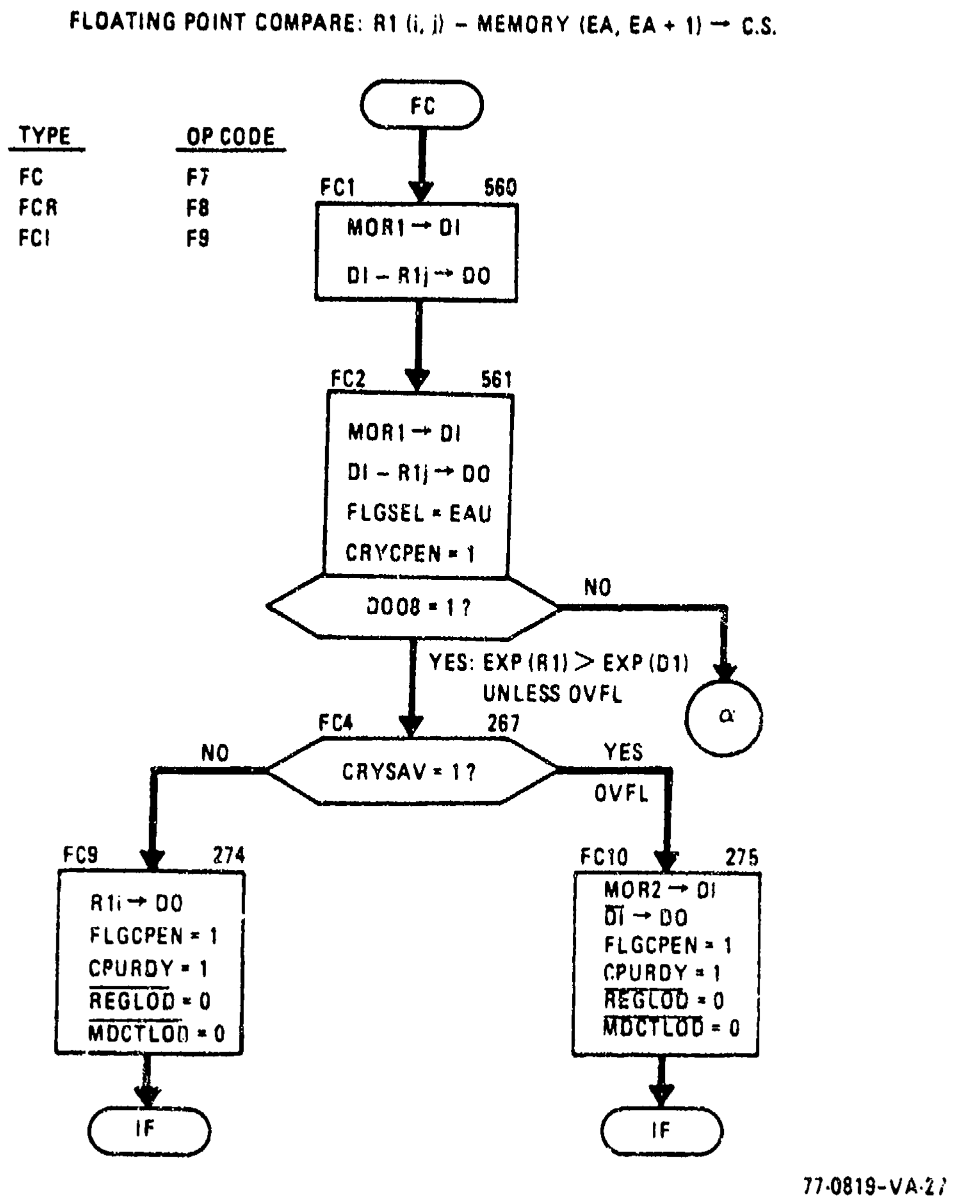


Figure 30 . FCR Instruction

MNEMONIC: M **OP CODE:** C0

SHORT NAME: single precision multiply

FORMAT:

M	R1, ADDR		nonindexed
M	R1, ADDR, RX		indexed

1 1 0 0 0 0 0 0	R1	RX	ADDRESS FIELD

DESCRIPTION: The memory operand is multiplied by the content of register R1. The high order part of the product is retained in register R1: the lower order part of the product is retained in register R1+1.

The condition status, CS, is set based on the result. If RX is 0, then the 16-bit address field is used as a memory address to obtain the memory operand. If RX is nonzero, then the content of register RX is added to the 16-bit address field and the resulting sum is used as a memory address to obtain the memory operand.

REGISTERS AFFECTED: R1, R1+1, CS

TIMING: 4.0

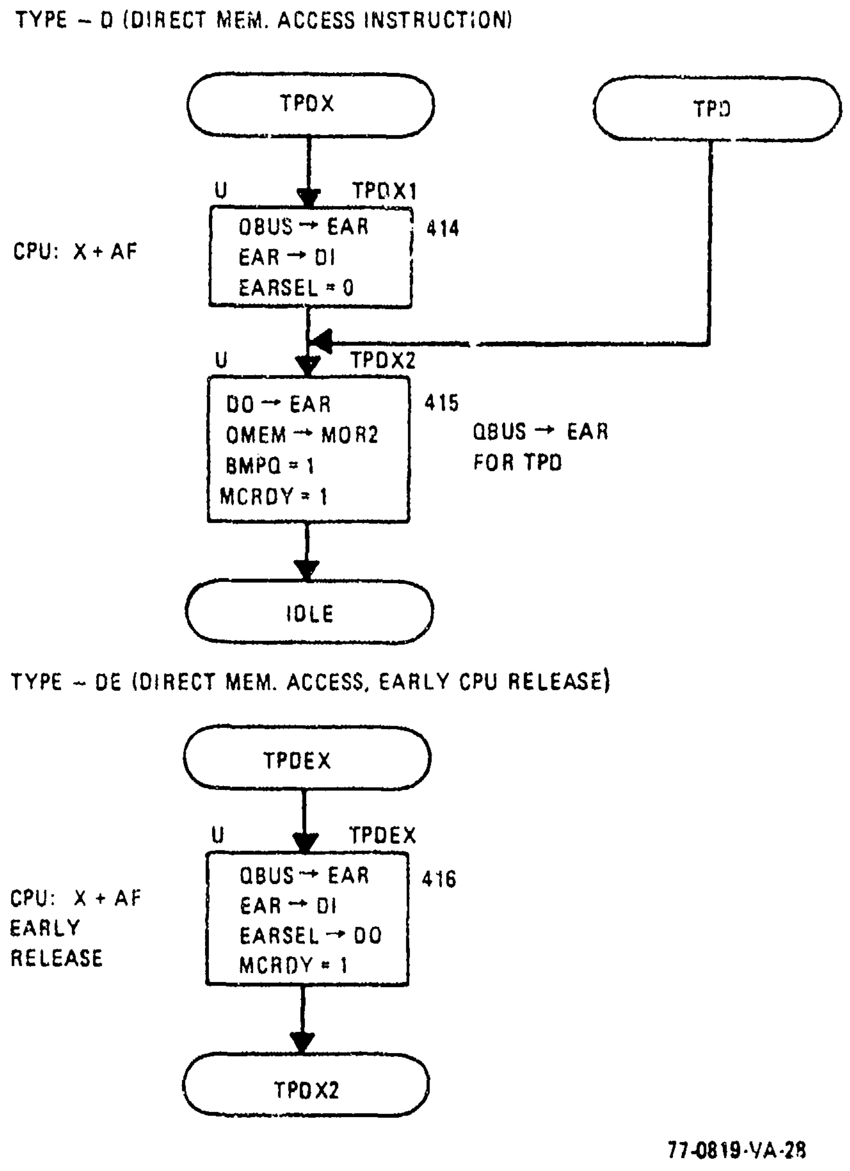


Figure 31. TYPE - D (Direct Memory Access Instruction)

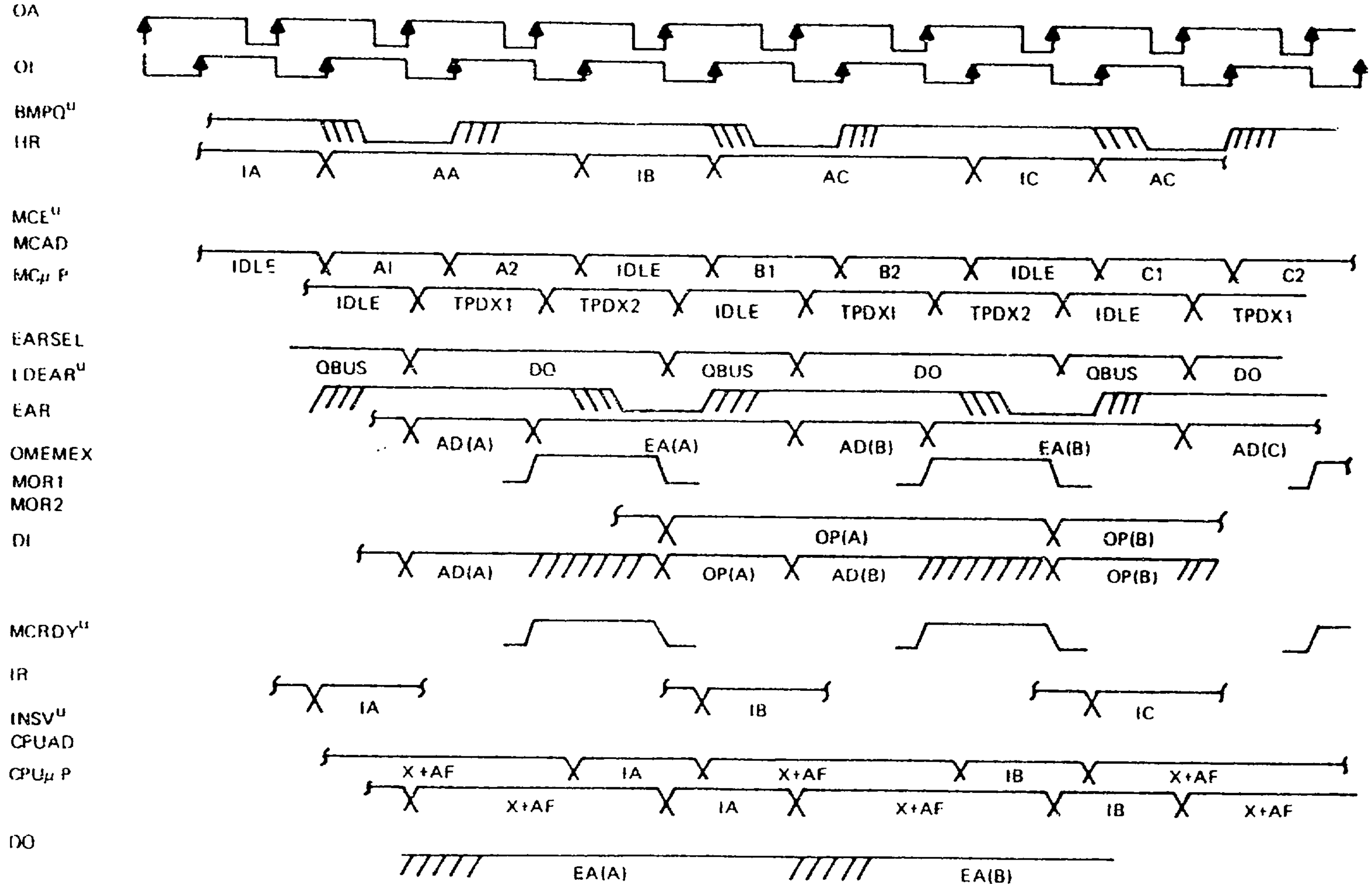
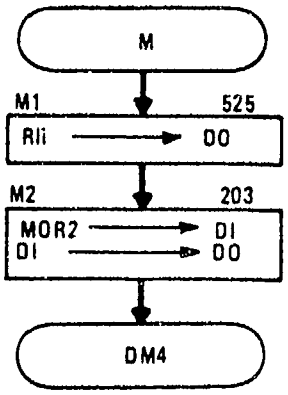


Figure 32. M Timing Diagram

SINGLE PRECISION MULTIPLY: R1i * MEMORY → R1i

TYPE	OP CODE
M	CO
MI	C2
MB	10,11,12,13
MIM	C3

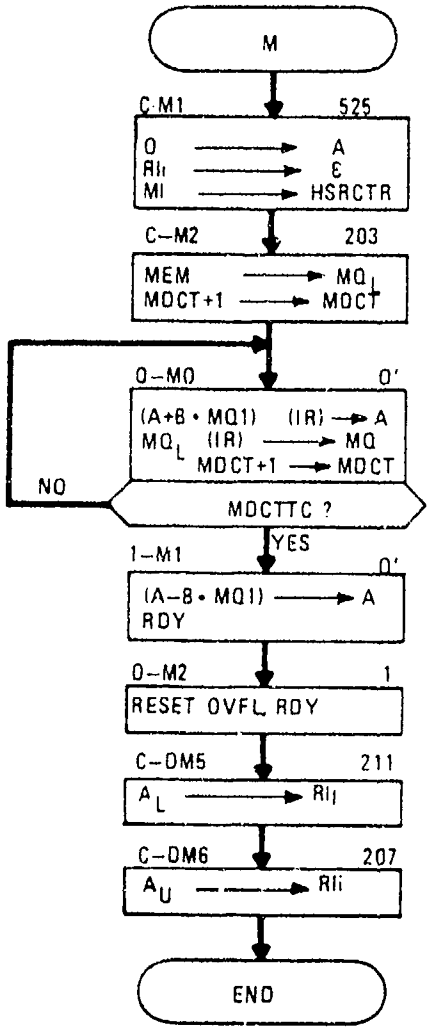


77-0819-VA-29

Figure 33 . M Instruction

FRACTIONAL MULTIPLY

M	CU
MI	C2
MIM	C3
MB	10,11,12,13



77-0819-VA-30

Figure 34 . M Instruction

MNEMONIC: MR OP CODE: C1

SHORT NAME: single precision multiply, register-to-register

FORMAT: MR R1, R2

1	1	0	0	0	0	0	1	R1	R2

DESCRIPTION: The content of register R2 is multiplied by the content of register R1 and the product is retained in register R1 and R1+1. The condition status, CS, is set based on the result.

REGISTERS AFFECTED: R1, R1+1, CS

TIMING: [illegible]

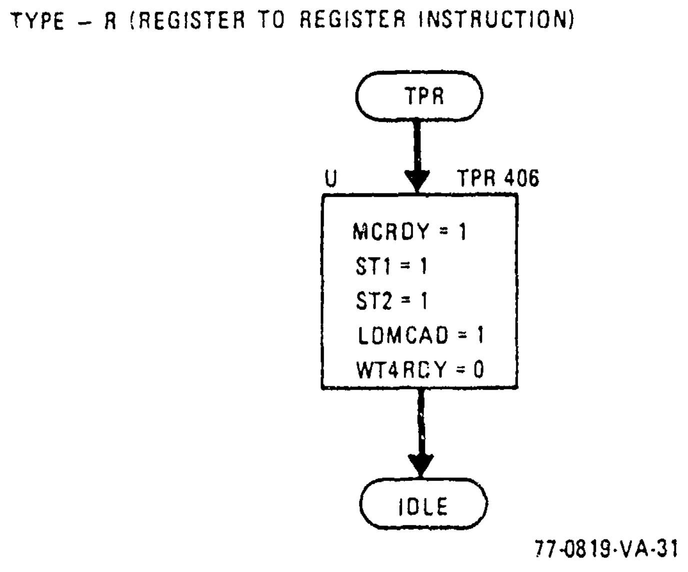


Figure 33. TYPE - R (Register to Register Instruction)

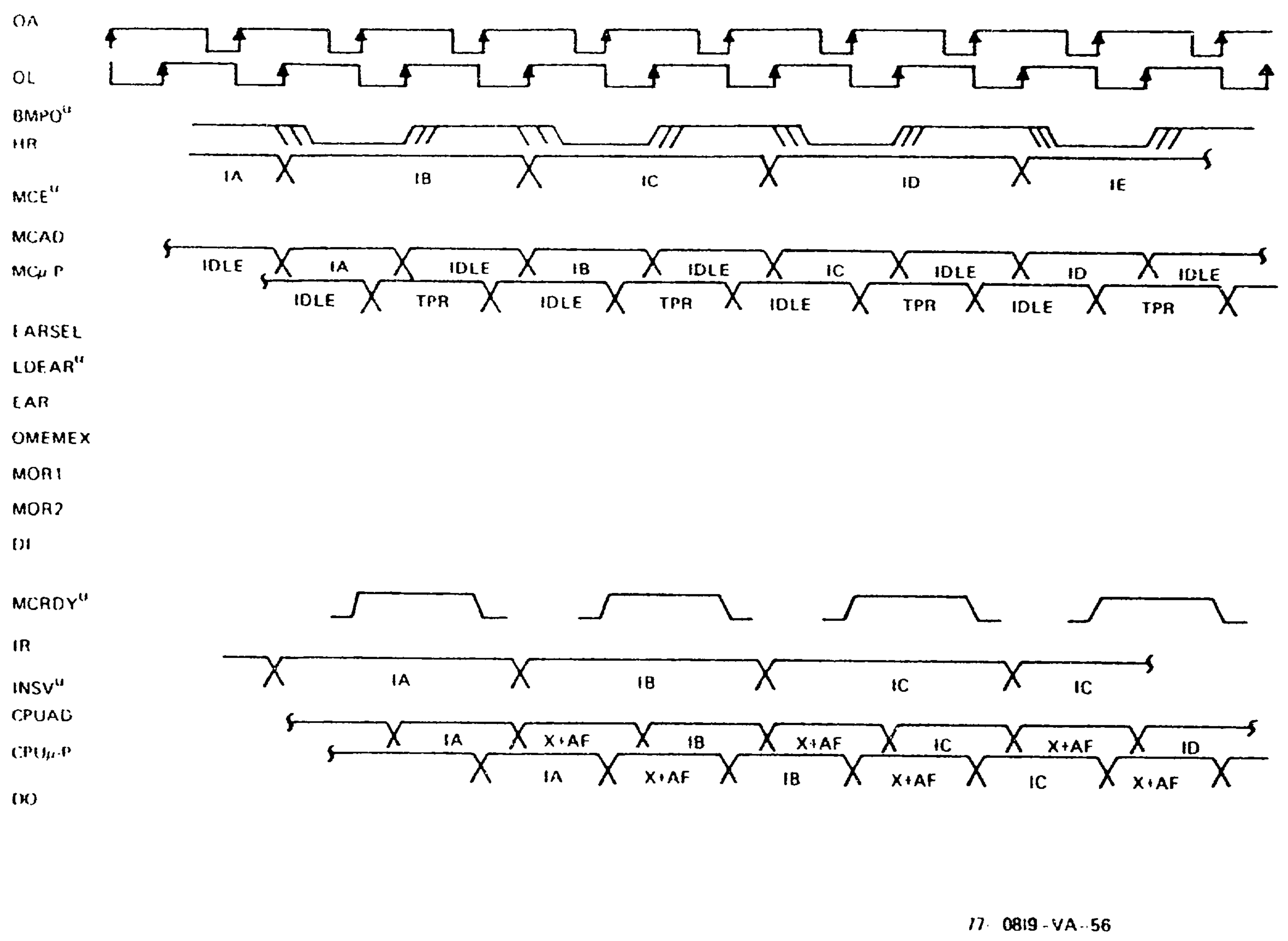
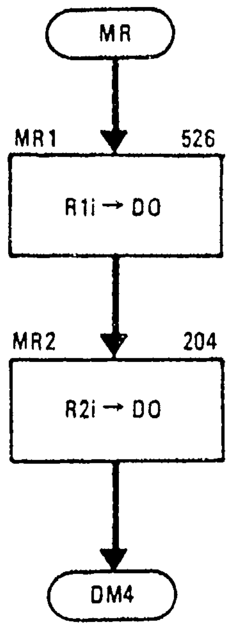


Figure 36. MR Timing Diagram

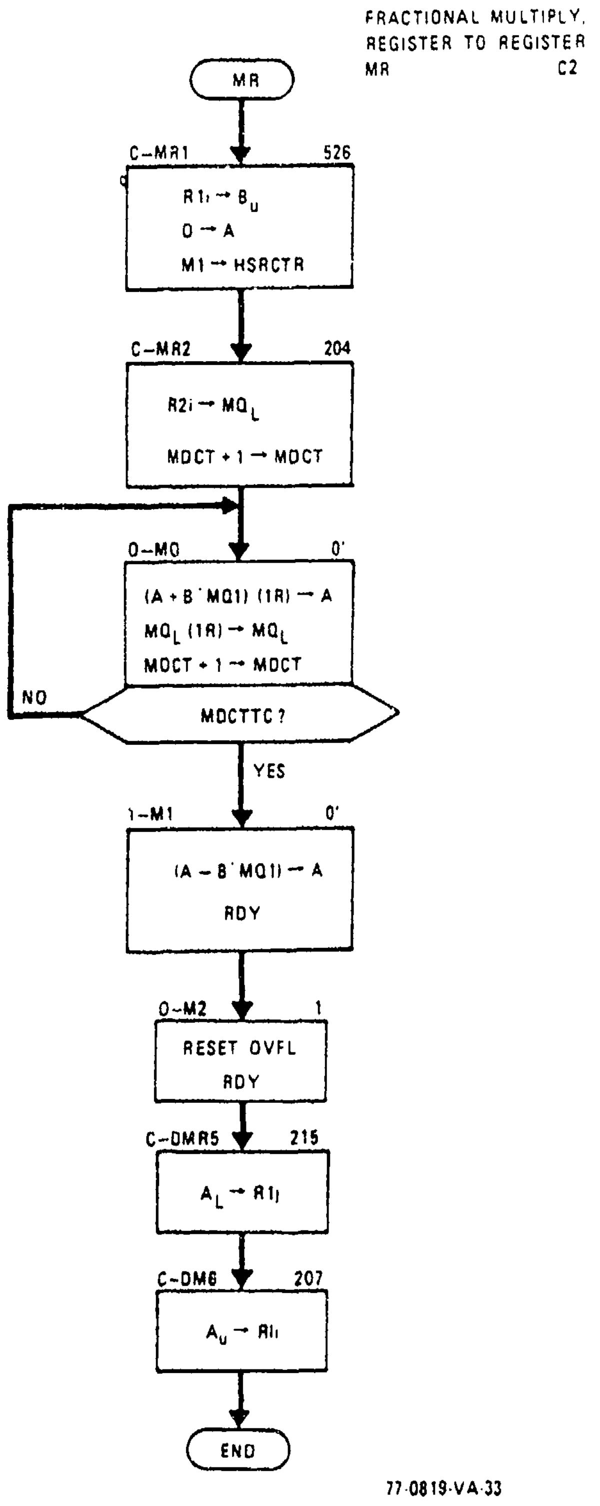
SINGLE PRECISION MULTIPLY: REGISTER TO REGISTER: R1i – R2i   R1i

TYPE	OP CODE
MR	C1



77-0819-VA-32

Figure 37. MR Instruction



15

Figure 38. MR Instruction

MNEMONIC: MI **OP CODE:** C2

SHORT NAME: single precision multiply indirect

FORMAT:

MI	R1, ADDR	nonindexed
MI	R1, ADDR, RX	indexed

1	1	0	0	0	0	1	0	R1	RX	ADDRESS FIELD

DESCRIPTION: The memory operand is multiplied by the content of register R1. The product is retained in register R1 and R1+1. The condition status, CS, is set based on the result.

If RX is 0, then the 16-bit address field is used to fetch a memory address. This memory address is used to obtain the memory operand. If RX is nonzero, then the 16-bit address field is used to fetch an address. The content of register RX is added to the fetched address and the resulting sum is used as a memory address to obtain the memory operand.

REGISTERS AFFECTED: R1, CS

TIMING: 5.0

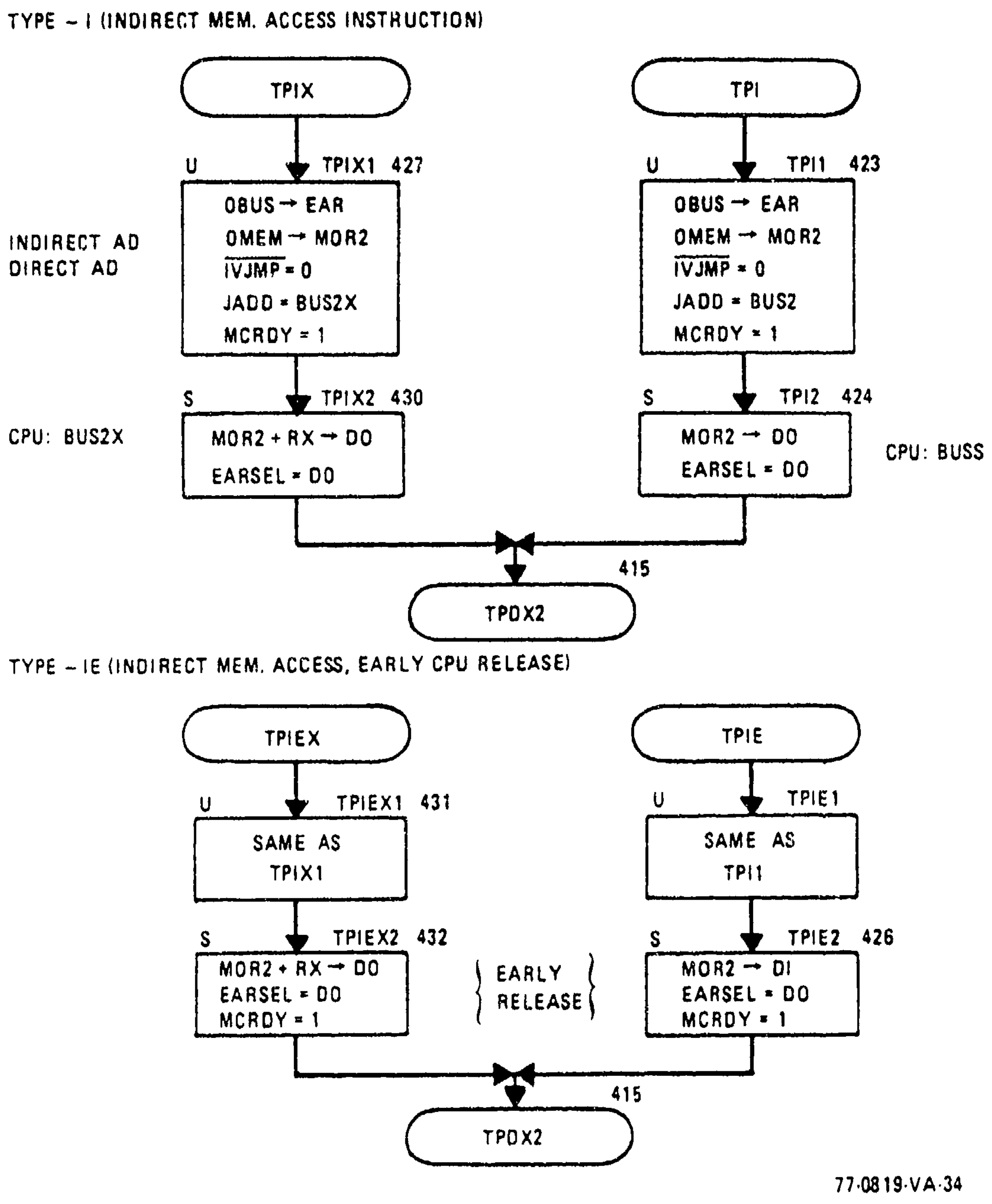


Figure 39. Type - I (Indirect Memory Access Instruction)

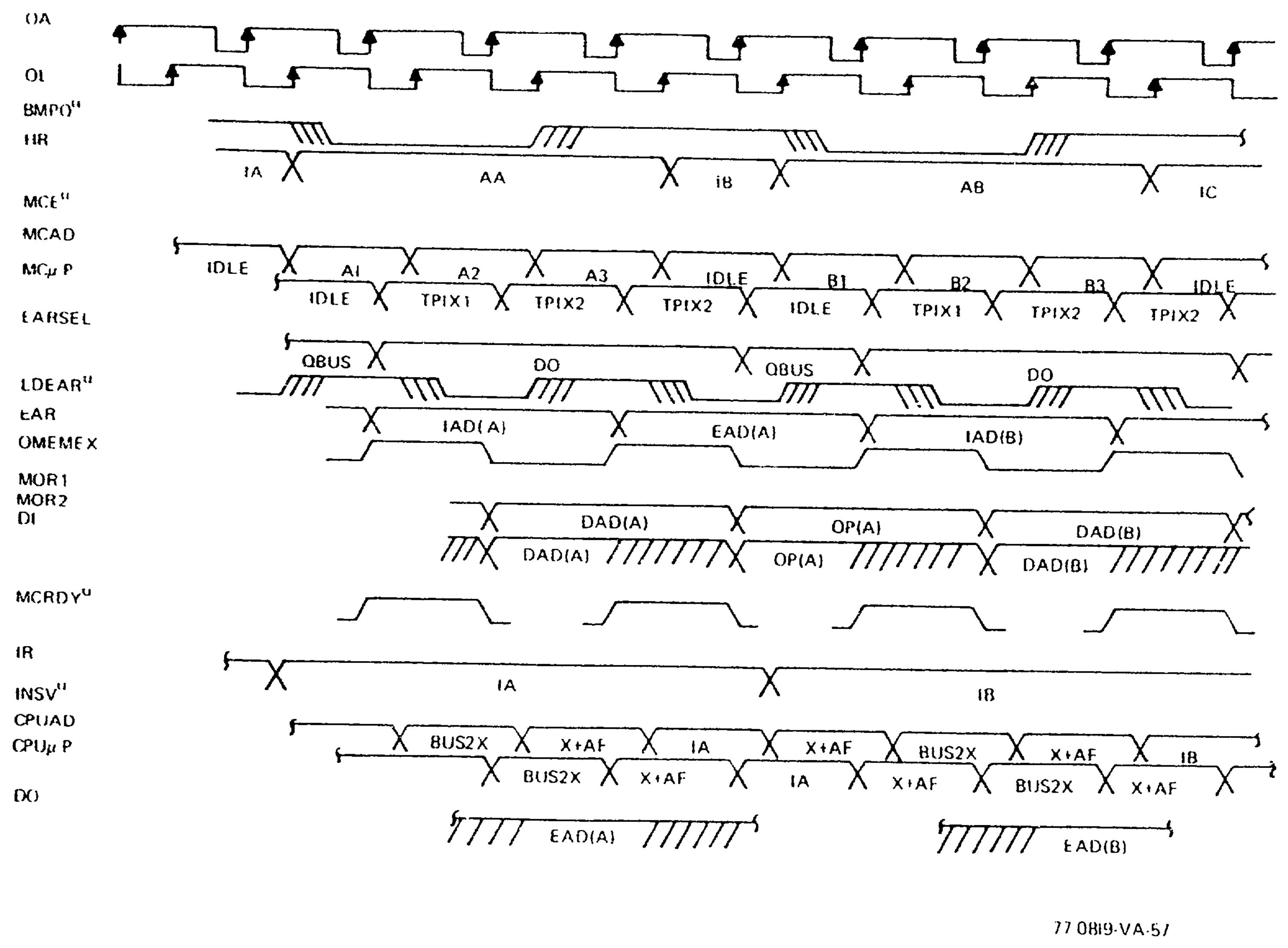
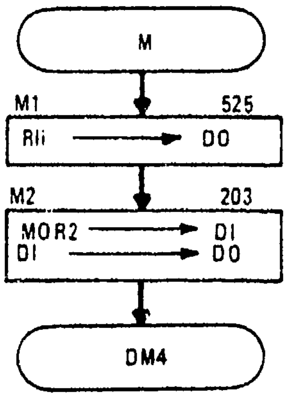


Figure 40. MI Timing Diagram

SINGLE PRECISION MULTIPLY: R1i * MEMORY → R1i

TYPE	OP CODE
M	C0
MI	C2
MB	10,11,12,13
MIM	C3

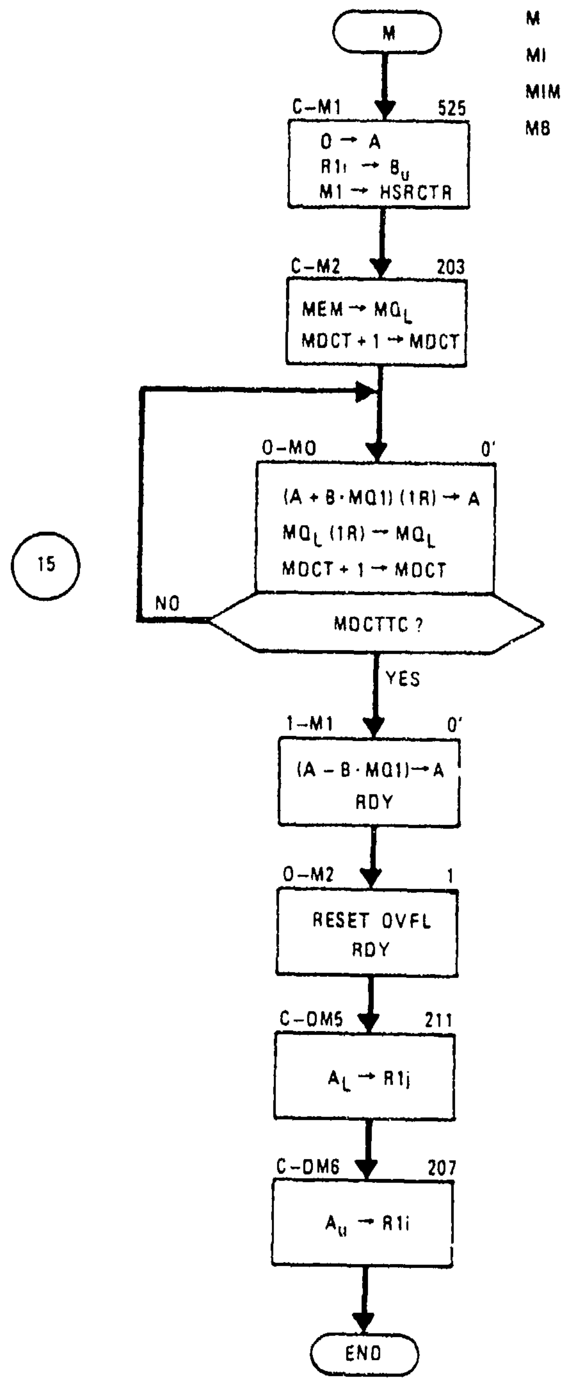


77-0819-VA-29

Figure 41 . MI Instruction

FRACTIONAL MULTIPLY

M	C0
MI	C2
MIM	C3
MB	10, 11, 12, 13



77-0819-VA-35

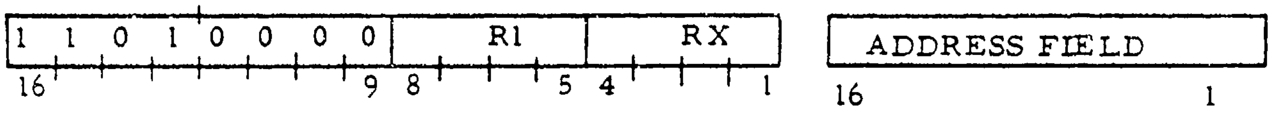
Figure 42 . MI Instruction

MNEMONIC: D　　　　　　　　　　OP CODE: D0

SHORT NAME: single precision divide

FORMAT:

D	Rl, ADDR		nonindexed
D	Rl, ADDR, RX		indexed



DESCRIPTION: The content of register Rl and Rl+1 is divided by the memory operand. The quotient is retained in register Rl and the remainder is retained in register Rl + 1. Overflow occurs if the magnitude of the number in storage is equal or less than the magnitude in register Rl. The condition status, CS, is set based on the result in register Rl and overflow. If RX is 0, then the 16-bit address field is used as a memory address to obtain the memory operand. If RX is nonzero, then the content of register RX is added to the 16-bit address field and the resulting sum is used as a memory address to obtain the memory operand. Rl must be even.

REGISTERS AFFECTED: Rl, Rl + l, CS

TIMING: 4.2

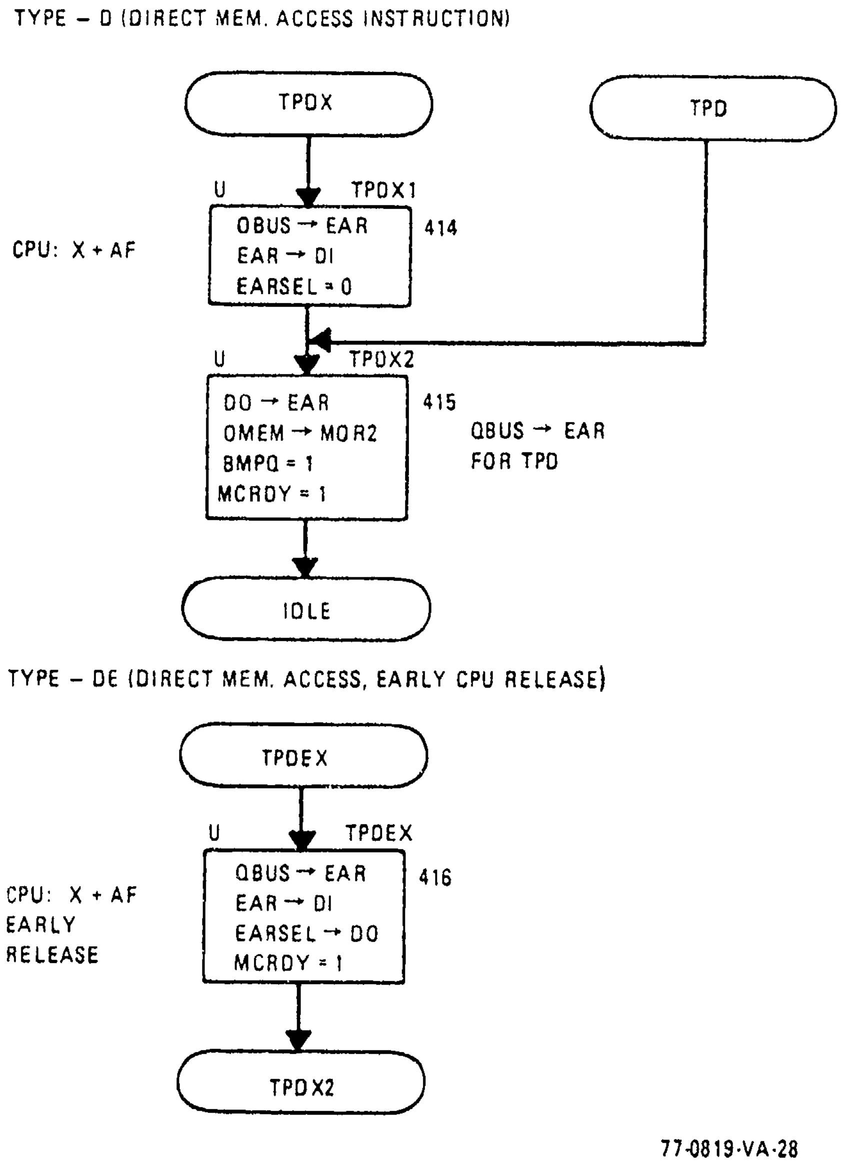


Figure 43. Type - D (Direct Memory Access Instruction)

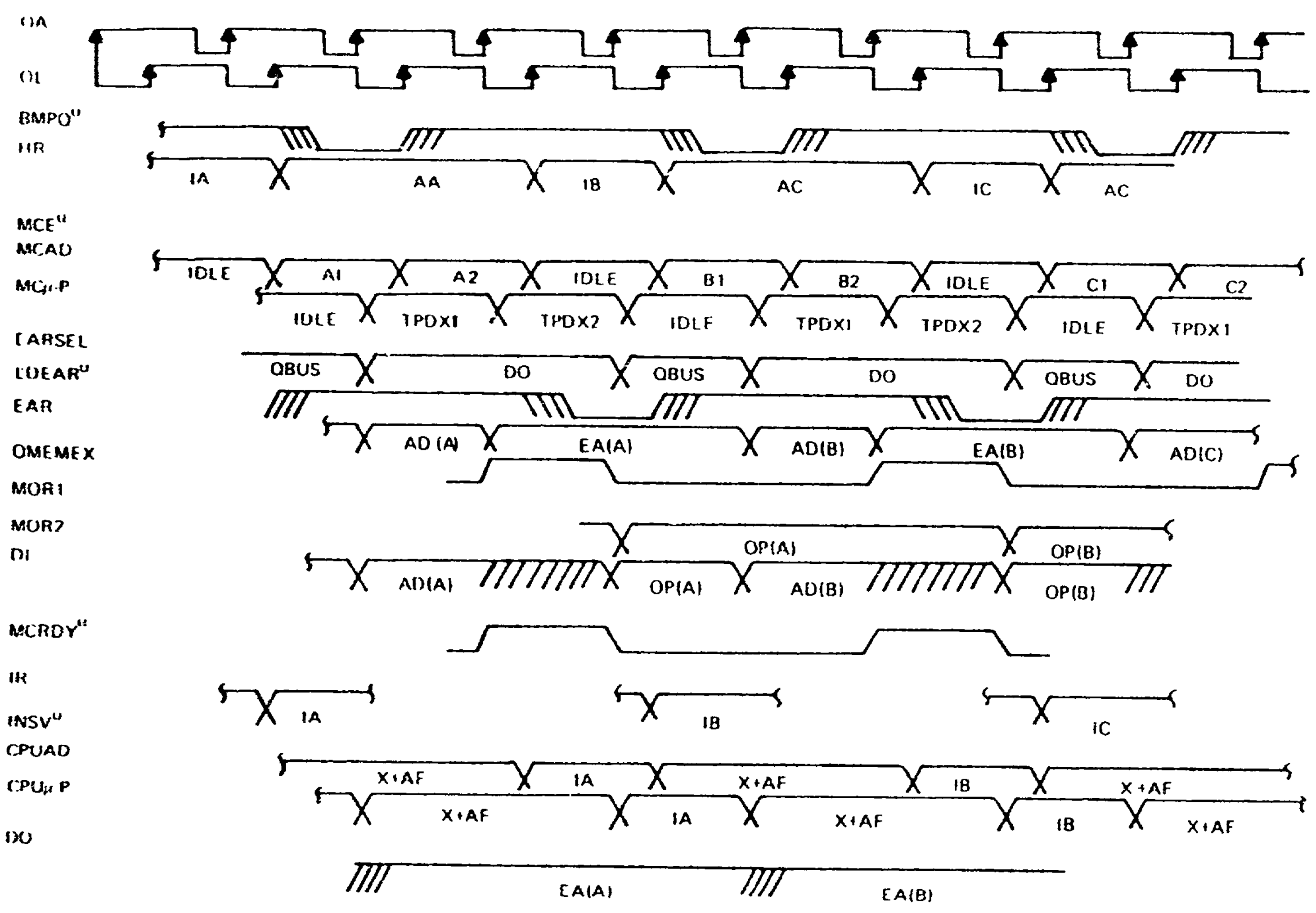
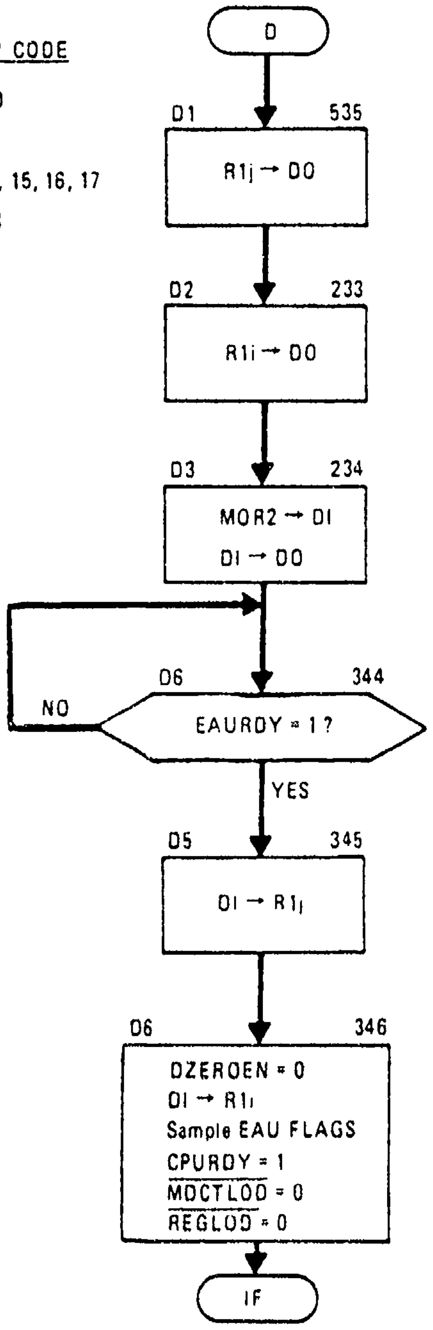


Figure 44. D Timing Diagram

SINGLE PRECISION DIVIDE, REGISTER TO MEMORY: (R1i, R1j) ÷ MEMORY → (R1i, R1j)
(IN EAU) (Q, R)

TYPE	OP CODE
D	D0
DI	D2
DB	14, 15, 16, 17
DIM	D3



77-0819-VA-36

Figure 45. D Instruction

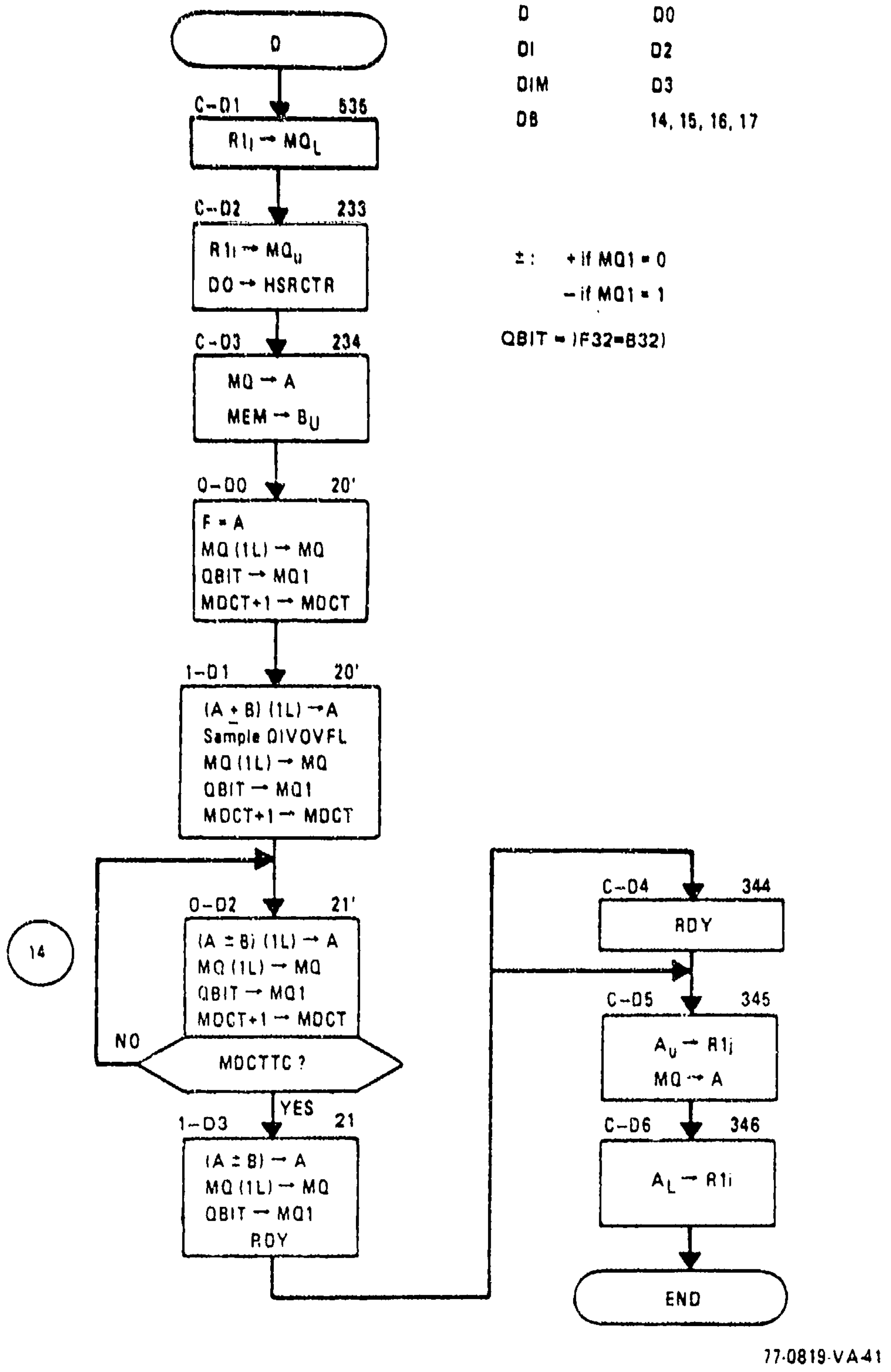


Figure 46. D Instruction

MNEMONIC: DR OP CODE: D1

SHORT NAME: single precision divide, register-to-register

FORMAT: DR R1, R2

1	1	0	1	0	0	0	1	R1	R2

DESCRIPTION: The content of registers R1 and R1 + 1 is divided by the content of register R2. The quotient is retained in register R1 and the remainder is retained in register R1 plus one. The condition status, CS, is set based on the result in register R1 and overflow. R1 must be even.

REGISTERS AFFECTED: R1, R1+1, CS

TIMING: 4.0

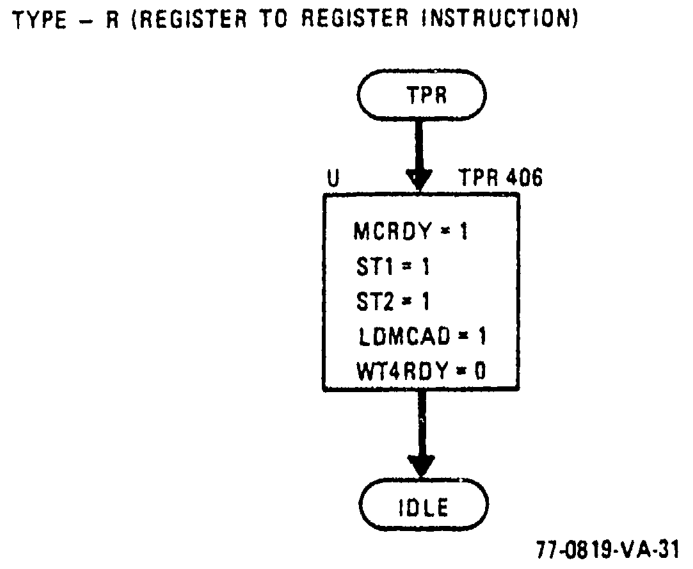


Figure 47 . Type - R (Register to Register Instruction)

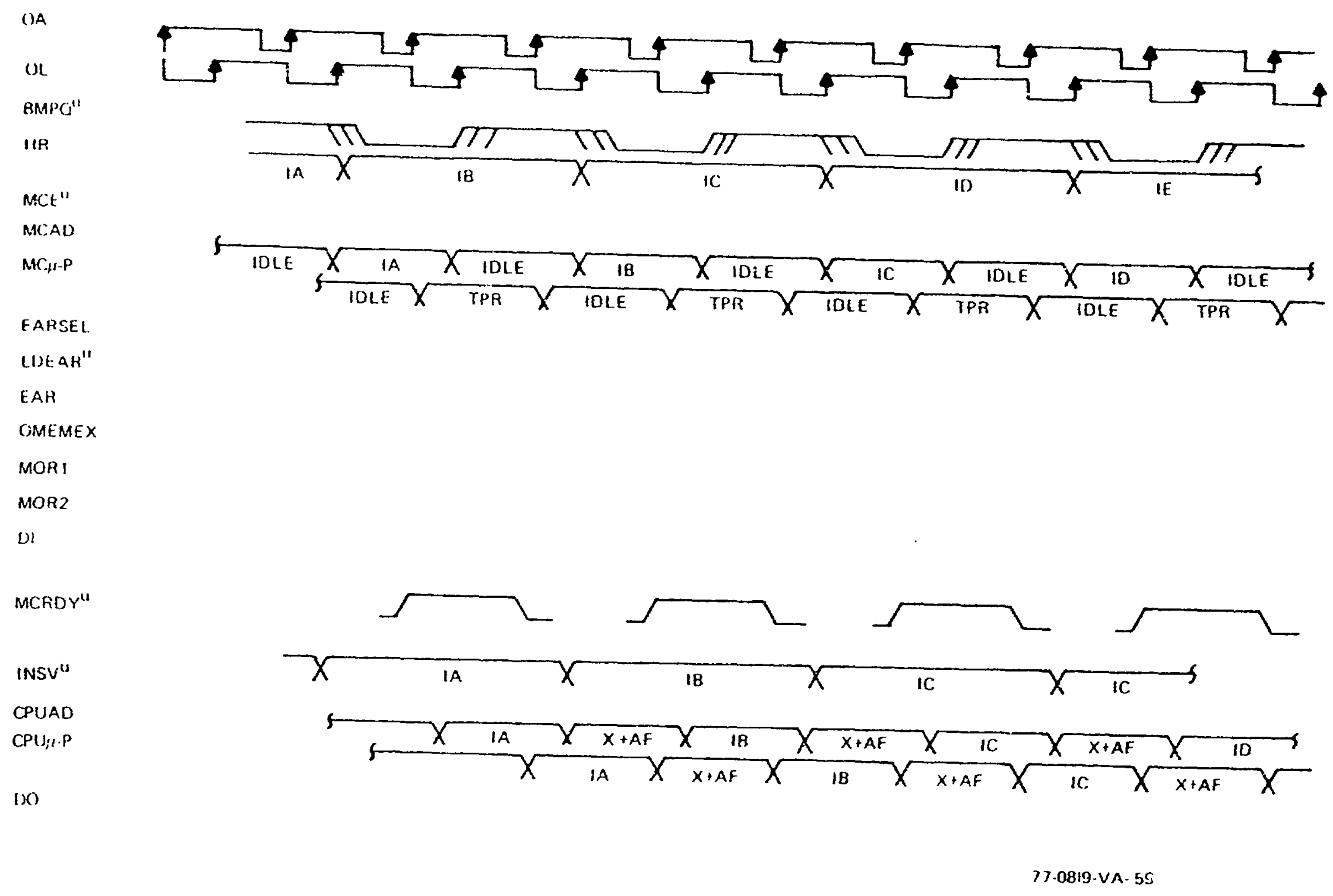
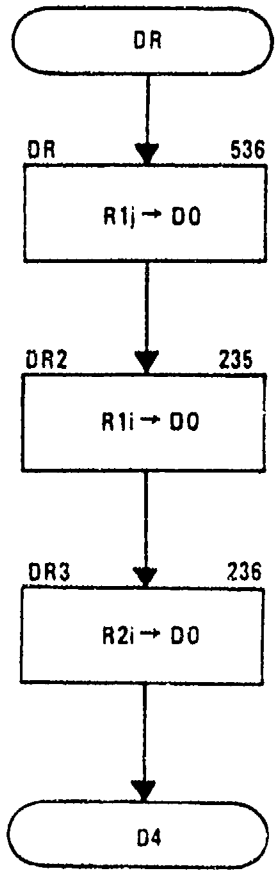


Figure 48 . DR Timing Diagram

SINGLE PRECISION DIVIDE, REGISTER TO REGISTER: $(R1i, R1j) \div R2i \rightarrow \underset{Q,R}{(R1i, R1j)}$

TYPE	OP CODE
DR	D1



77-0819-VA-38

Figure 49 . DR Instruction

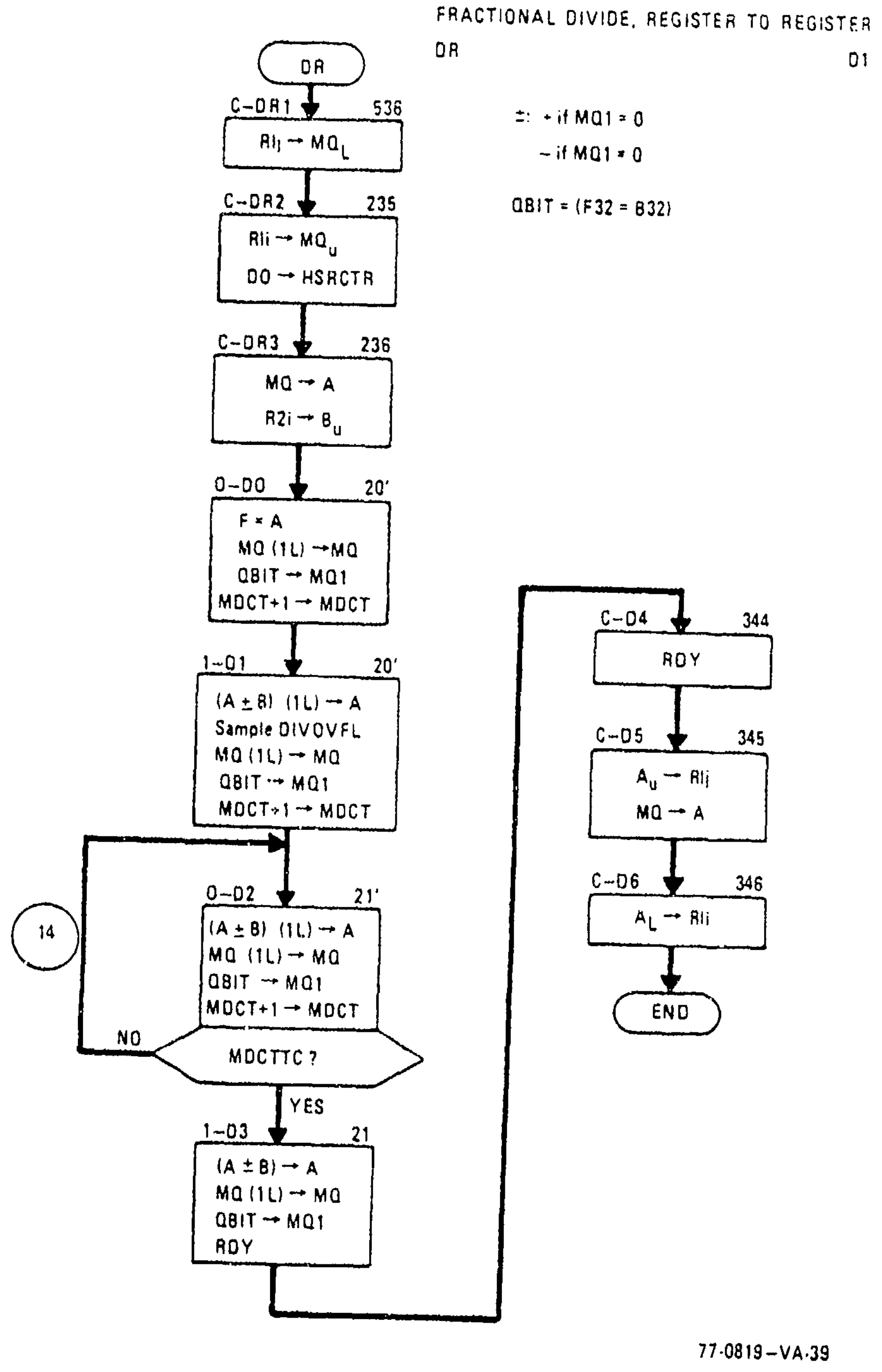


Figure 50 . DR Instruction

MNEMONIC: DI OP CODE: D2

SHORT NAME: single precision divide indirect

FORMAT:	DI	R1, ADDR	nonindexed
	DI	R1, ADDR, RX	indexed

1 1 0 1 0 0 1 0	R1	RX	ADDRESS FIELD

DESCRIPTION: The content of register R1 and R1 + 1 is divided by the memory operand. The quotient is retained in register R1 and the remainder is retained in register R1 + 1. The condition status, CS, is set based on the result in register R1 and overflow. R1 must be even.

If RX is 0, then the 16-bit address field is used to fetch memory address. This memory address is used to obtain the memory operand. If RX is nonzero, then the 16-bit address field is used to fetch an address. The content of register RX is added to the fetched address and the resulting is used as a memory address to obtain the memory operand.

REGISTERS AFFECTED: R1, R1 + 1, CS

TIMING: 5.2

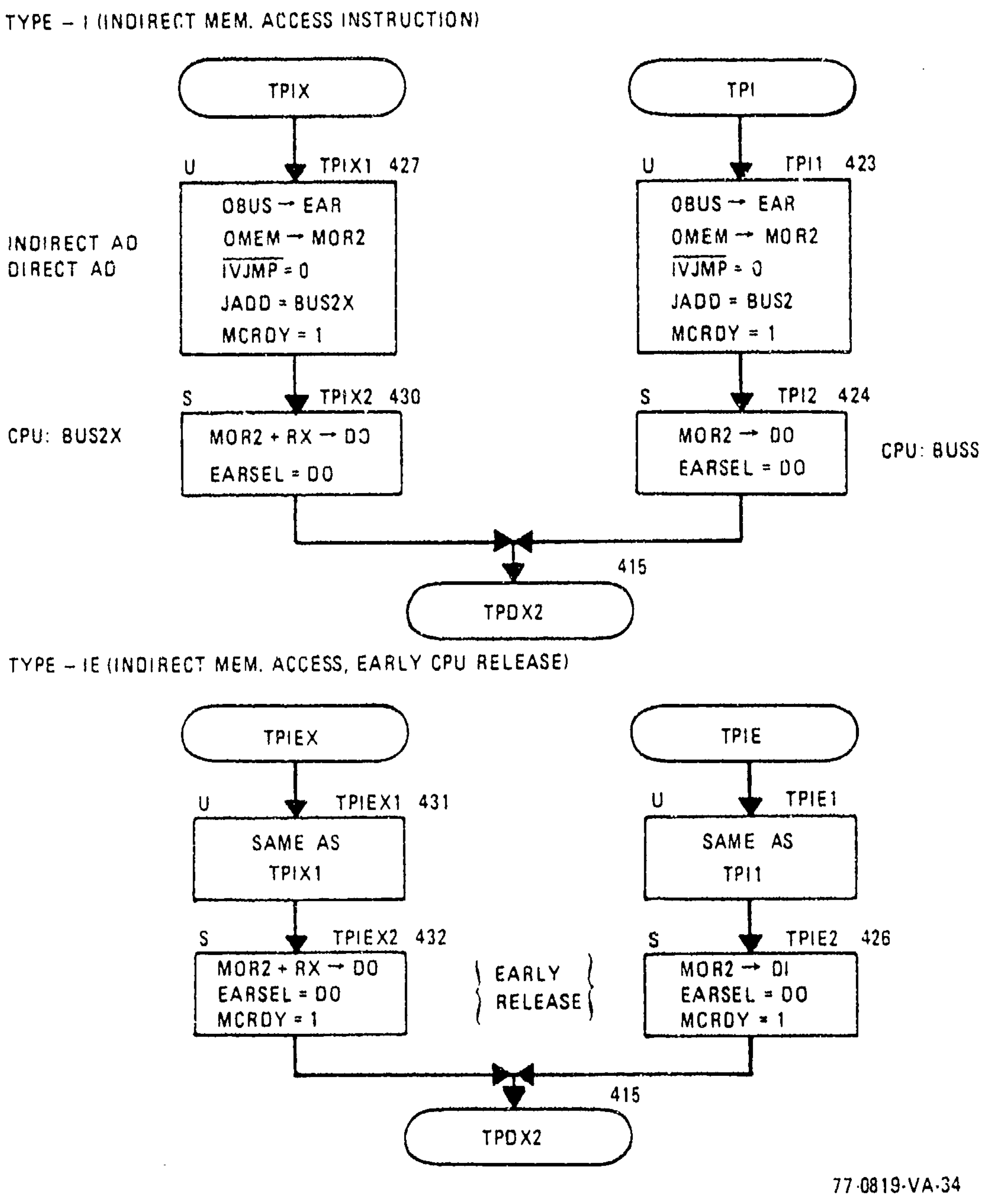


Figure 51 . Type - I (Indirect Memory Access Instruction)

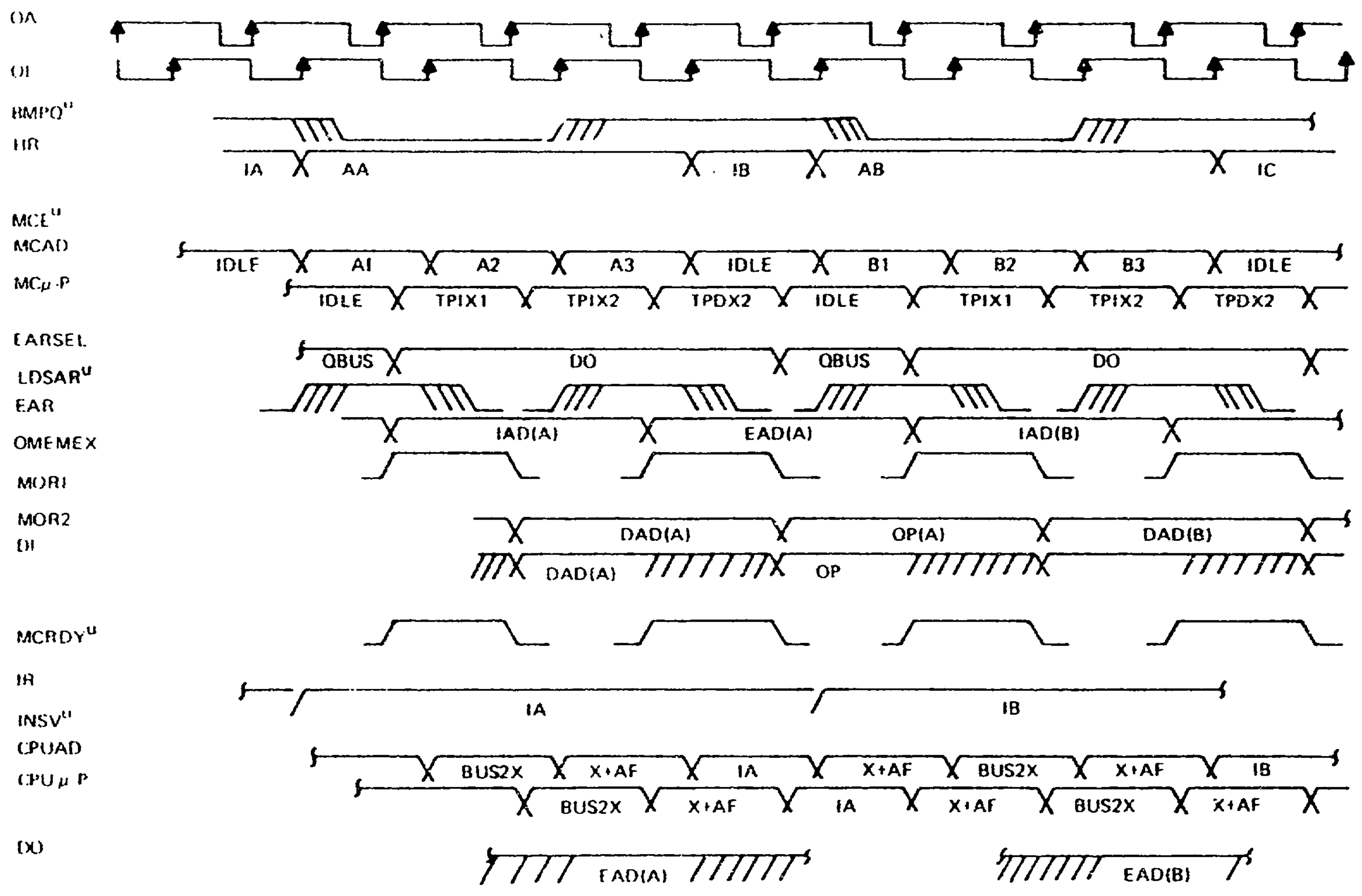
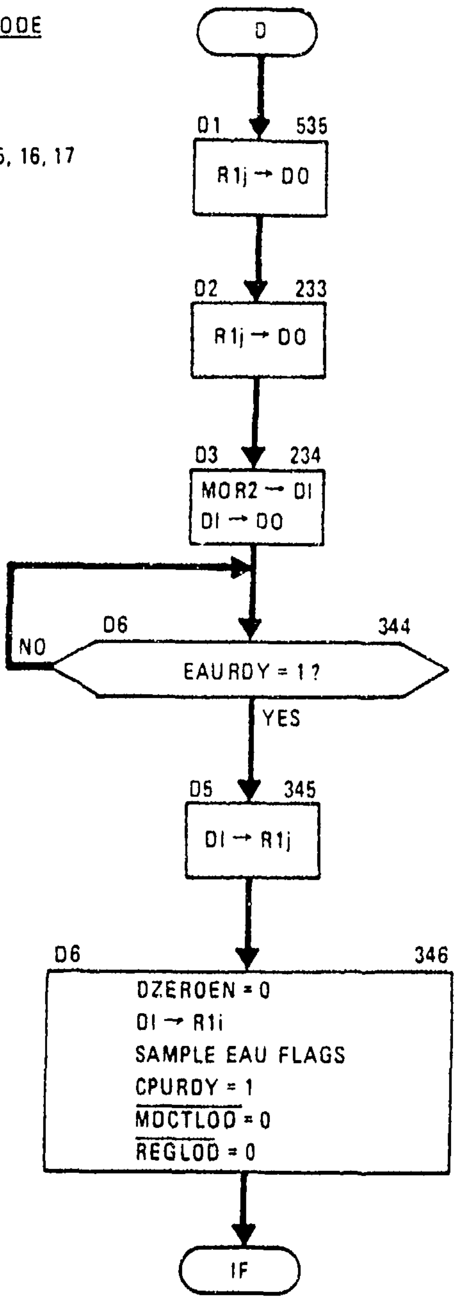


Figure 52. DI Timing Diagram

SINGLE PRECISION DIVIDE, REGISTER TO MEMORY: (R1i, R1j) ÷ MEMORY → (R1i, R1j)
(IN EAU) (Q, R)

TYPE	OP CODE
D	D0
DI	D2
DB	14, 15, 16, 17
DIM	D3



77-0819-VA-40

Figure 53 . DI Instruction

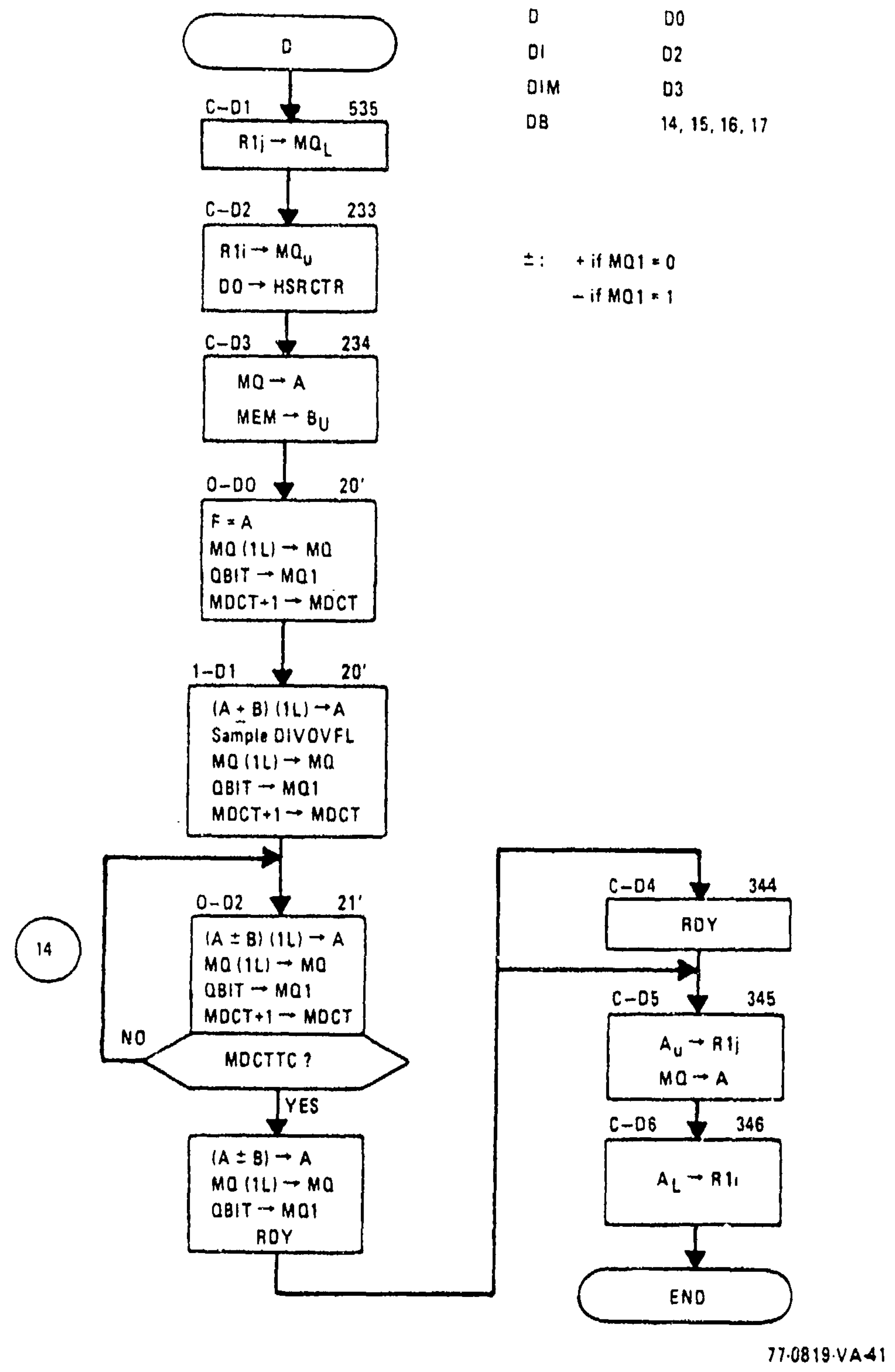


Figure 54 . DI Instruction

MNEMONIC: DABS **OP CODE:** AC

SHORT NAME: double precision absolute value register to register

FORMAT: DABS R1, R2
DABS R1

1	0	1	0	1	1	0	0	R1	R2

DESCRIPTION: If the sign bit of register R2 is a one, then double precision negate register R2, R2 + 1 and place result in R1 and R1 + 1, otherwise place R2, R2 + 1 in R1, R1 + 1, respectively. R1 and R2 must be even. R1 may equal R2.

REGISTERS AFFECTED: R1, R1 + 1, CS

TIMING: 1.6

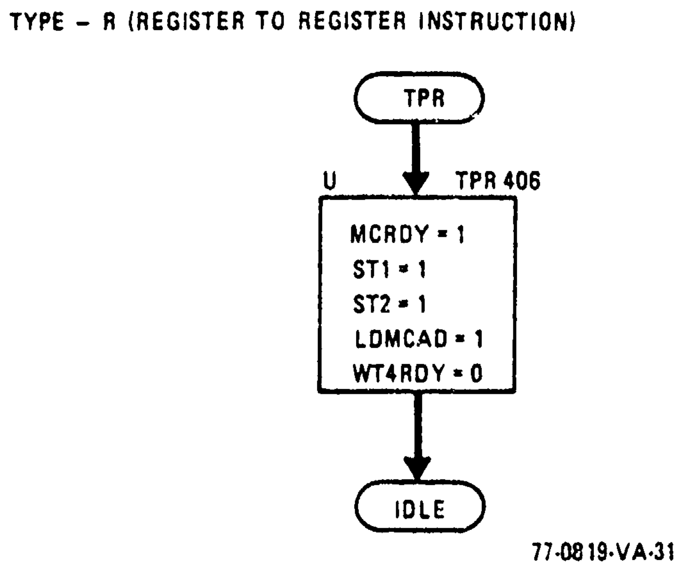


Figure 55. Type - R (Register to Register Instruction)

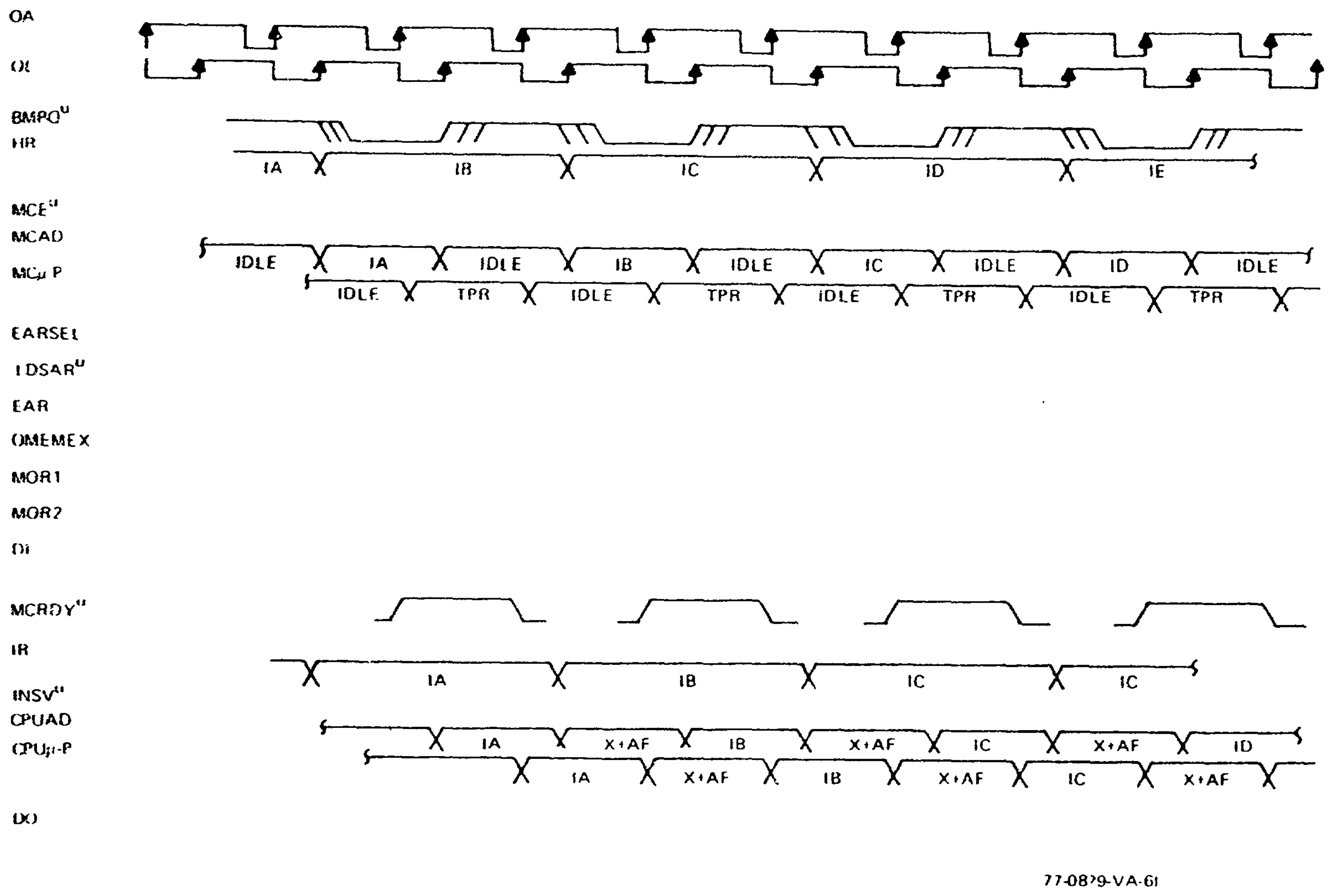


Figure 56. DABS Timing Diagram

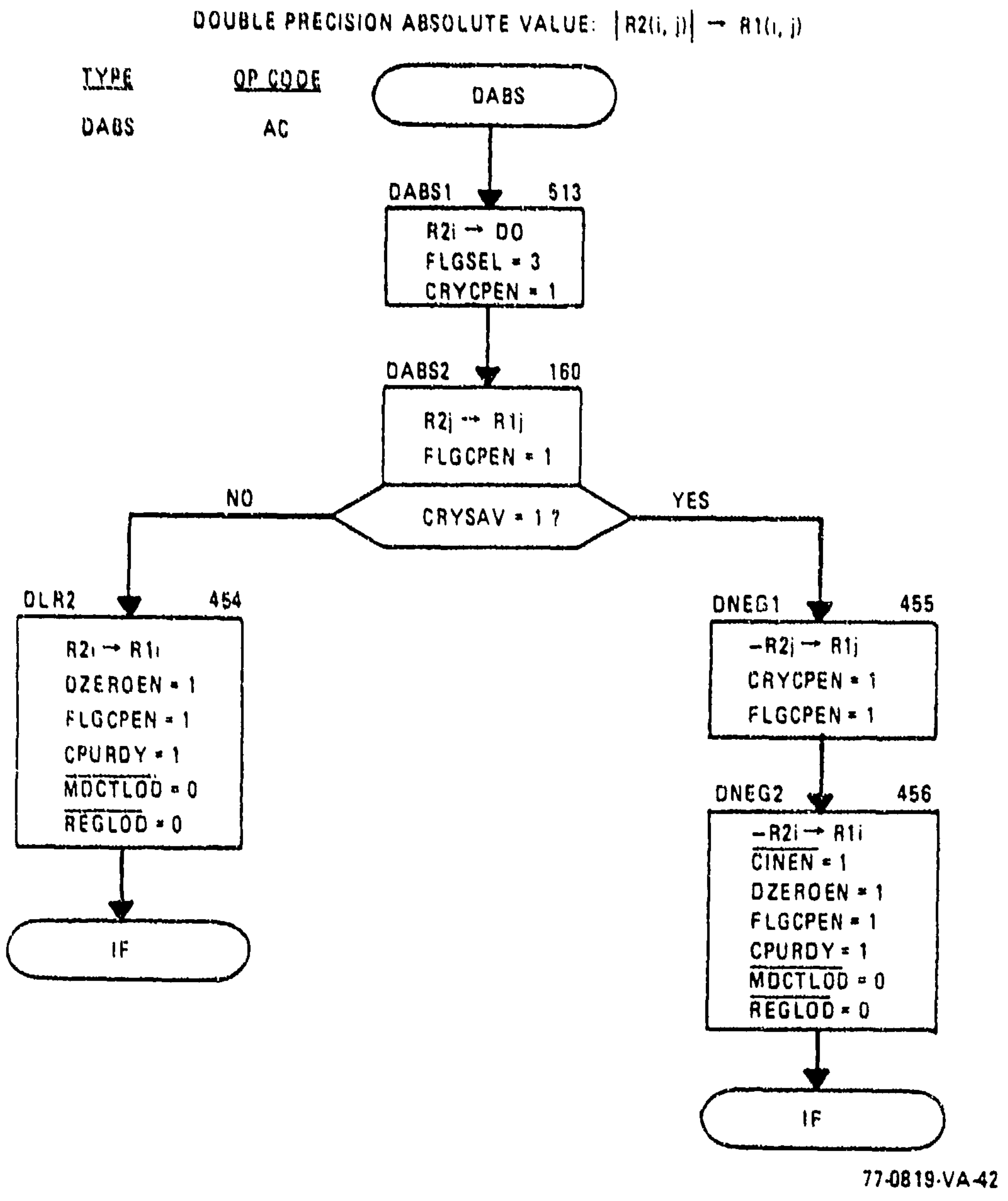


Figure 57 . DABS Instruction

MNEMONIC: DNEG **OP CODE:** BC

SHORT NAME: negate double precision register

FORMAT: DNEG R1, R2
DNEG R1

1	0	1	1	1	1	0	0	R1	R2

DESCRIPTION: The content of register R2 and Register R2 + 1 is negated. The result, the negative of the original double precision number, is placed in R1 and R1 + 1. R2 may be equal to R1. The condition status, CS, is set based on the double precision result in registers R1 and R1 + 1 and overflow. R1 and R2 must be even.

REGISTERS AFFECTED: R1, R1 + 1, CS

TIMING: 1.4

TYPE – R (REGISTER TO REGISTER INSTRUCTION)

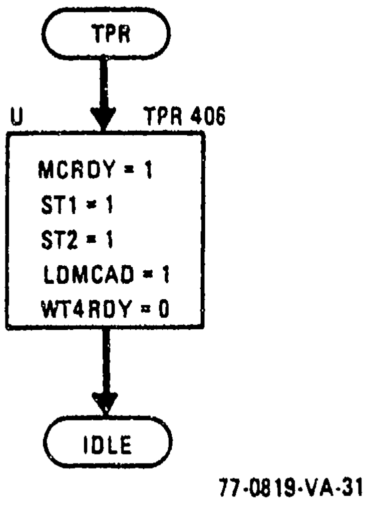


Figure 58 . Type - R (Register to Register Instruction)

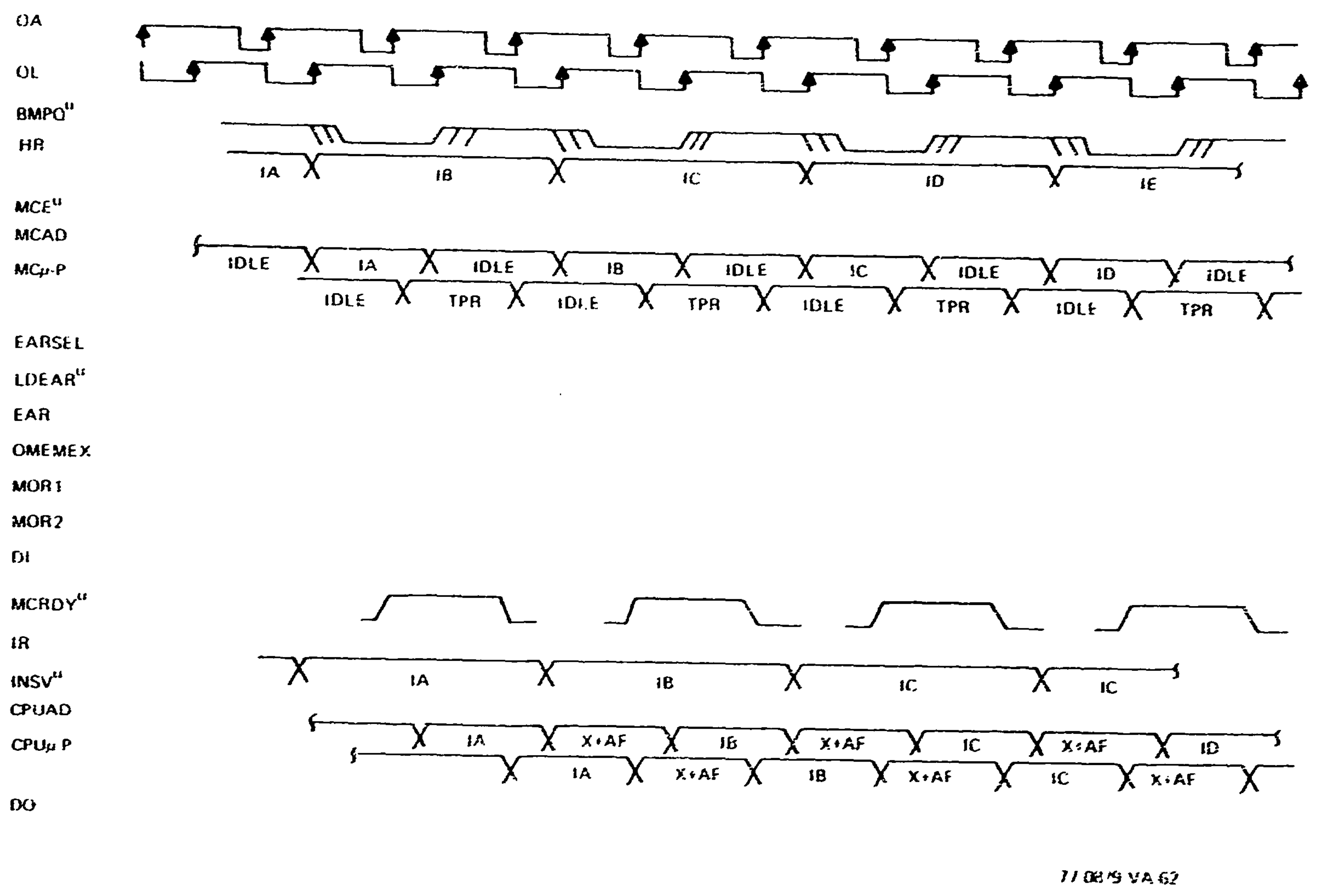


Figure 59. DNEG Timing Diagram

DOUBLE NEGATE: $-[R2(i,j)] \rightarrow R1(i,j)$

TYPE	OP CODE
DNEG	BC

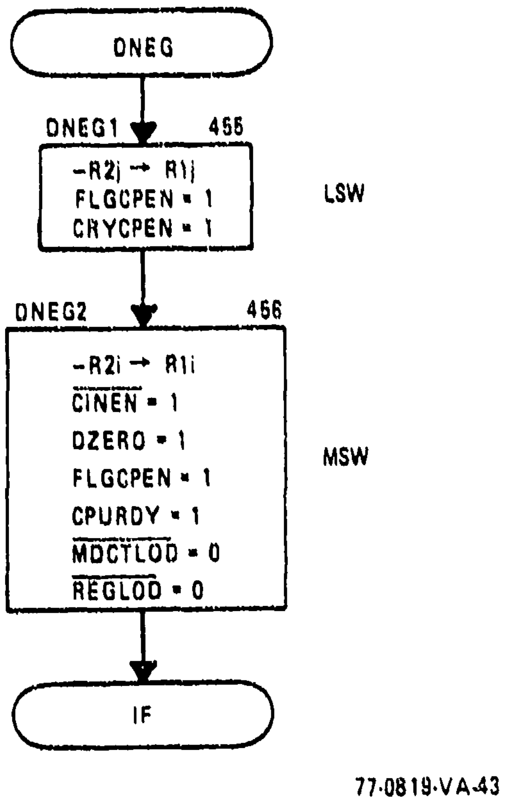


Figure 60 . DNEG Instruction

MNEMONIC: SRC OP CODE: 64

SHORT NAME: shift right cyclic

FORMAT: SRC R2, N

0	1	1	0	0	1	0	0	N-1	R2

DESCRIPTION: The content of register R2 is shifted right cyclically N positions. The field N-1 being zero represents a shift of 1 position. The field N-1 being 15 represents a shift of 16 positions. Bits shifted out of the least significant bit position enter the sign position. No bits are lost. The condition status, CS, is set based on the result in register R2. R2 may be any general register. The assembler subtracts 1 from the programs value of N and places N-1 in the 4 bit field.

Result in Register	Resulting Condition Status		
R2	Bits	Hex	JC Mnemonic
0	0010	2	EZ
sign bit = 1	0001	1	LZ
otherwise	0100	4	GZ

REGISTERS AFFECTED: R2, CS

TIMING: 1.4 + 0.4 per position

TYPE – R (REGISTER TO REGISTER INSTRUCTION)

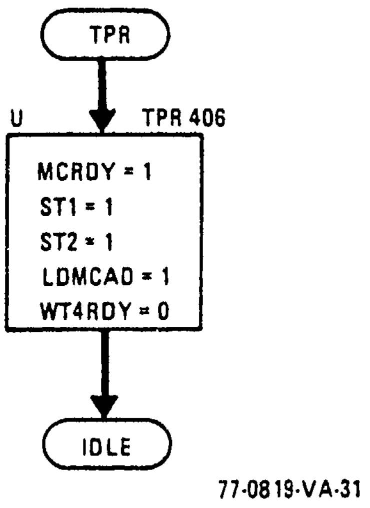


Figure 61 . Type - R (Register to Register Instruction)

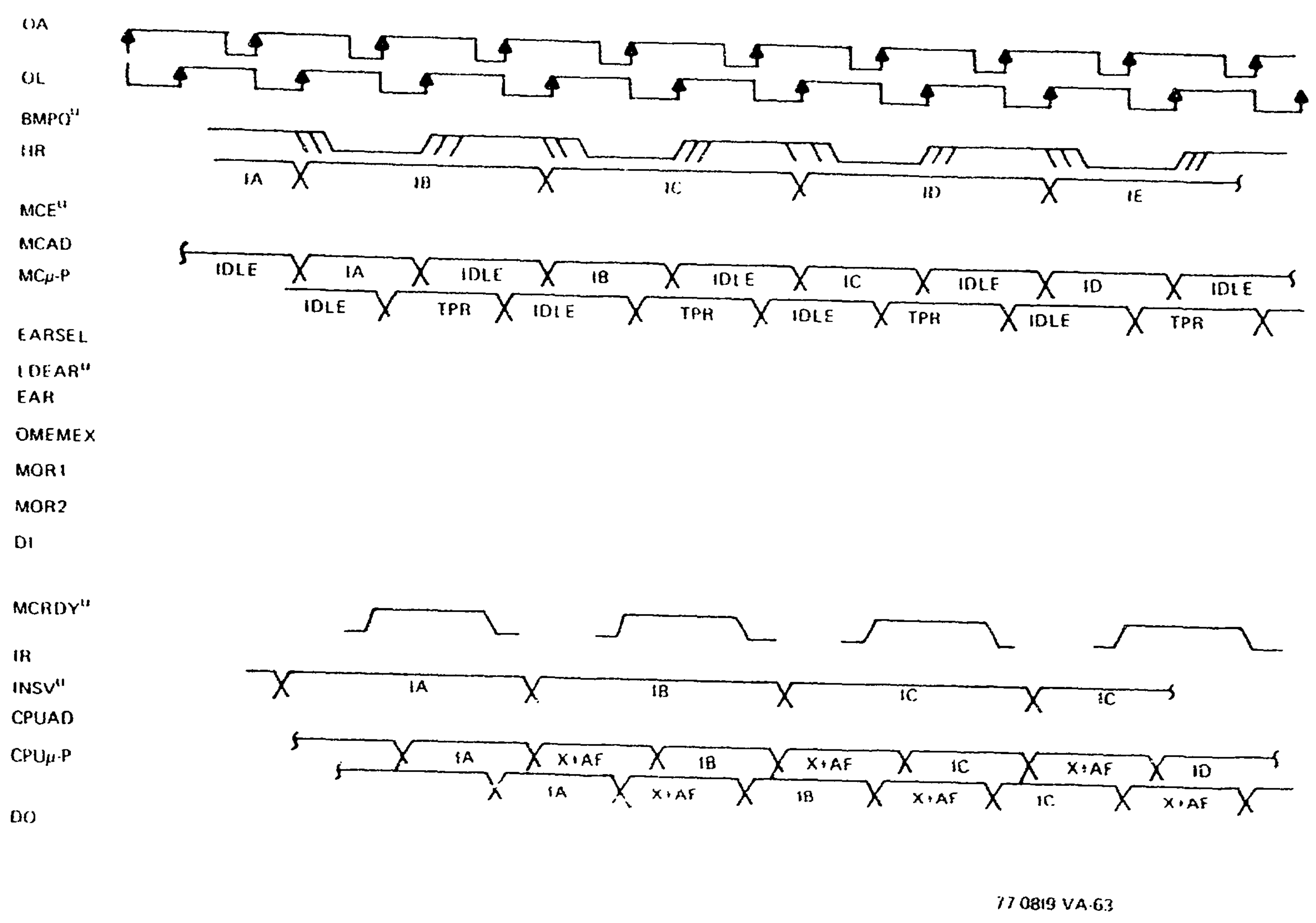
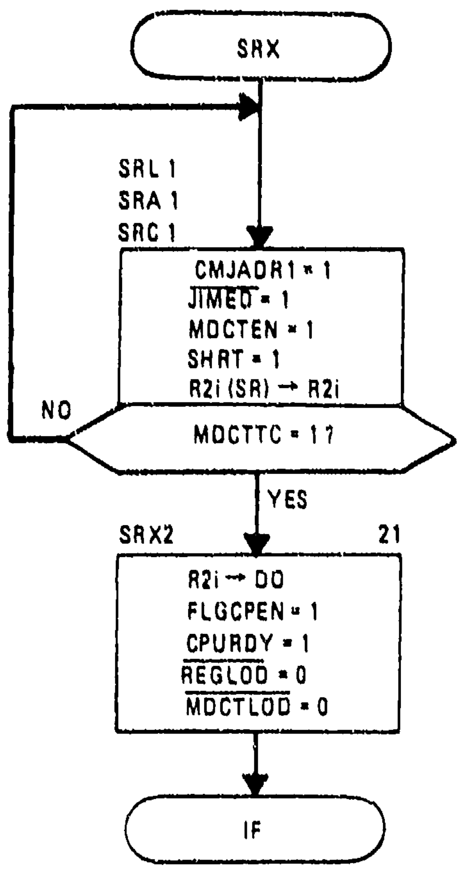


Figure 62. SRC Timing Diagram

SHIFT RIGHT: R2I → R2i (SHIFTED RIGHT N = 1 TIMES)

TYPE	OP CODE	
SRL	61	423
SRA	62	425
SRC	64	427


SRX
SRL 1
SRA 1
SRC 1
CMJADR1 = 1
$\overline{\text{JIMED}}$ = 1
MDCTEN = 1
SHRT = 1
R2i (SR) → R2i
NO
MDCTTC = 1?
YES
SRX2
21
R2i → DO
FLGCPEN = 1
$\overline{\text{CPURDY}}$ = 1
$\overline{\text{REGLOD}}$ = 0
$\overline{\text{MDCTLOD}}$ = 0
IF

77-0819-VA-44

Figure 63. SRC Instruction

MNEMONIC: DSLL **OP CODE:** 65

SHORT NAME: double shift left logical

FORMAT: DSLL R2, N

0	1	1	0	0	1	0	1	N-1	R2

DESCRIPTION: The content of registers R2 and R2 + 1 are shifted left logical N positions. The field N-1 being zero represents a shift of 1 position. The field N-1 being 15 represents a shift of 16 positions. Zeros enter the least significant position of register R2 + 1. Bits shifted out of the sign position of register R2 + 1 enter the least significant position of register R2. Bits shifted out of the sign position of register R2 are lost. The condition status, CS, is set based on the double precision result in registers R2 and R2 + 1. R2 must be even.

Result in Registers	Resulting Condition Status		
R2, R2 + 1	Bits	Hex	JC Mnemonic
both zero	0010	2	EZ
sign bit of R^2 = 1	0001	1	LZ
otherwise	0100	4	GZ

REGISTERS AFFECTED: R2, R2 + 1, CS

TIMING: 1.8 + 0.4 per position

TYPE – R (REGISTER TO REGISTER INSTRUCTION)

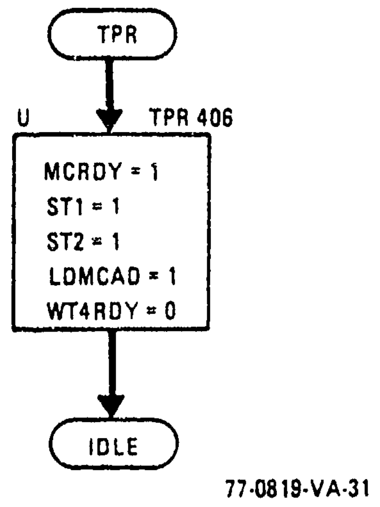


Figure 64 . Type - R (Register to Register Instruction)

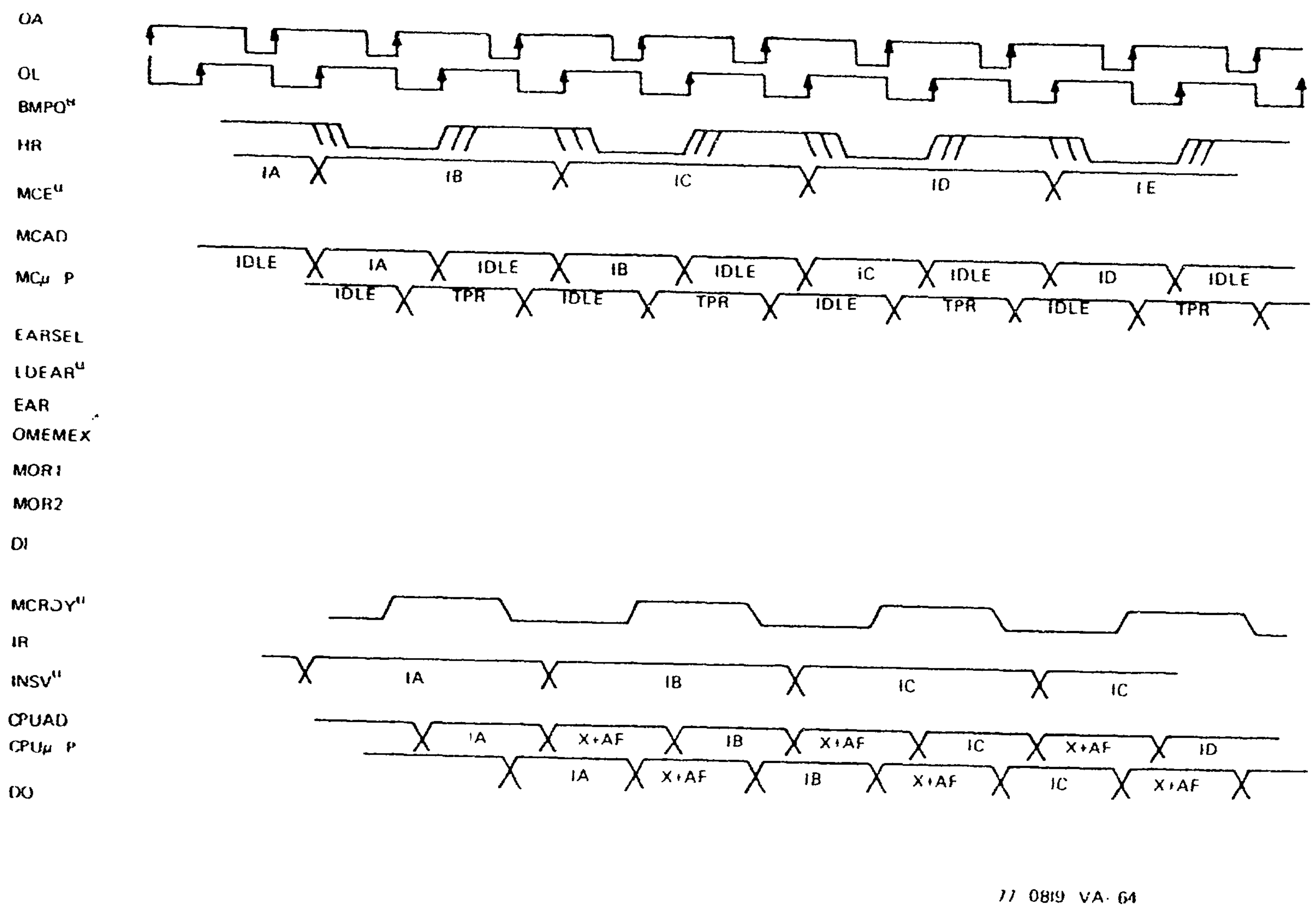
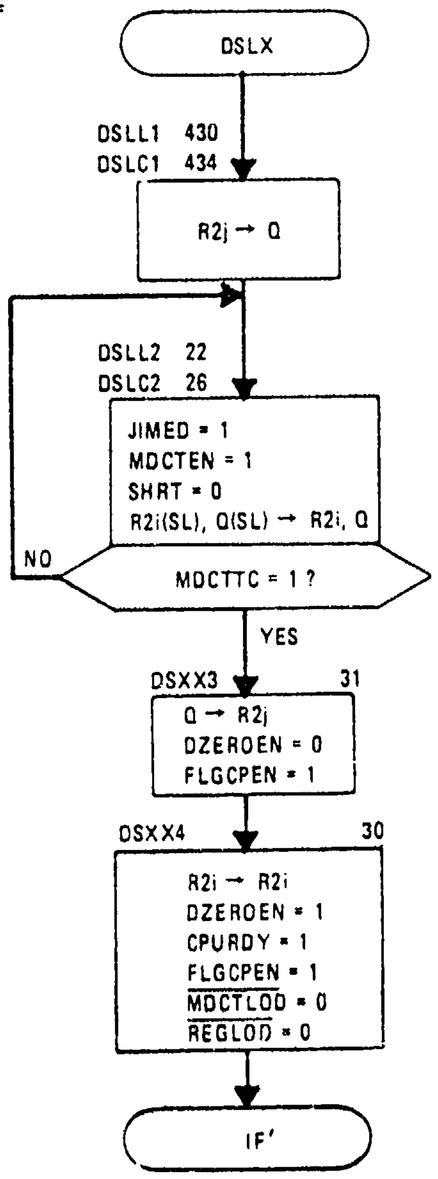


Figure 65 . DSLL Timing Diagram

DOUBLE PRECISION SHIFT LEFT: R2i, R2j ← R2i, R2j (SHIFTED N+ 1 TIMES)

TYPE	OP CODE
DSLL	65
DSLC	68



77-0819-VA-45

Figure 66. DSLL Instruction

MNEMONIC: DSRA OP CODE: 67

SHORT NAME: double shift right arithmetic

FORMAT: DSRA R2, N

0	1	1	0	0	1	1	1	N-1	R2

DESCRIPTION: The content of registers R2 and R2+1 is shifted right arithmetic N positions. The field N-1 being zero represents a shift of 1 position. The field N-1 being 15 represents a shift of 16 positions. The sign position of register R2 is not shifted. The sign bit is copied into the next position for each bit shifted. Bits leaving the least significant position of register R2 enter the sign position of register R2+1. Bits leaving the least significant position of register R2 + 1 are lost. The condition status, CS, is set based on the double precision result in registers R2 and R2 +1. R2 must be even.

Result in Registers	Resulting Condition Status		
R2, R2+1	Bits	Hex	JC Mnemonic
both zero	0010	2	EZ
sign bit of R2 = 1	0001	1	LZ
otherwise	0100	4	GZ

REGISTERS AFFECTED: R2, R2+1, CS

TIMING: 1.8 +0.4 per position

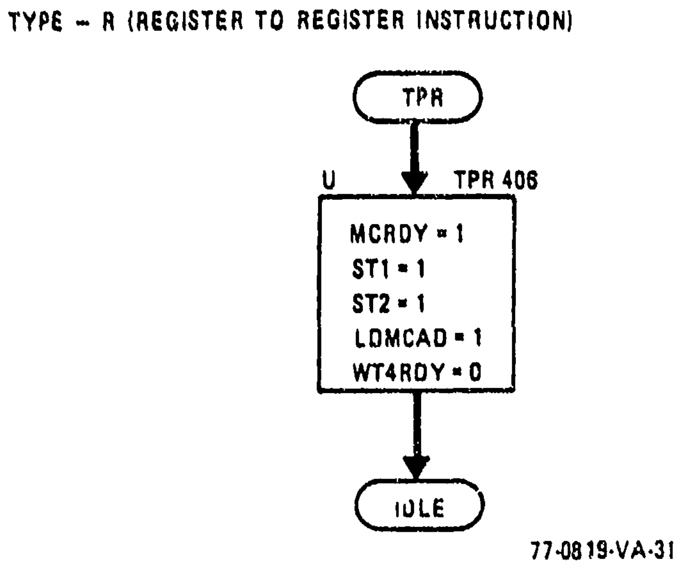


Figure 67. Type - R (Register to Register Instruction)

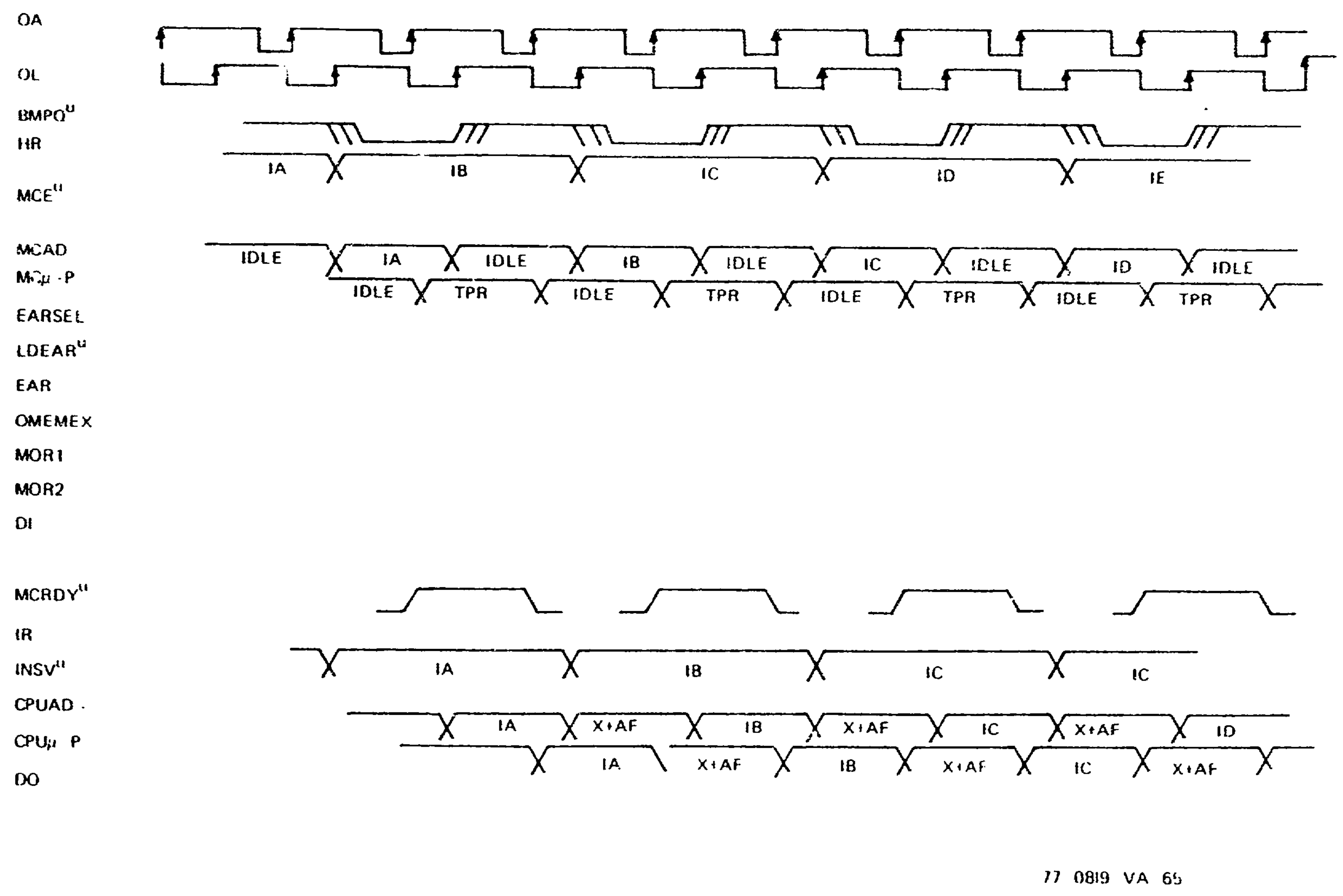
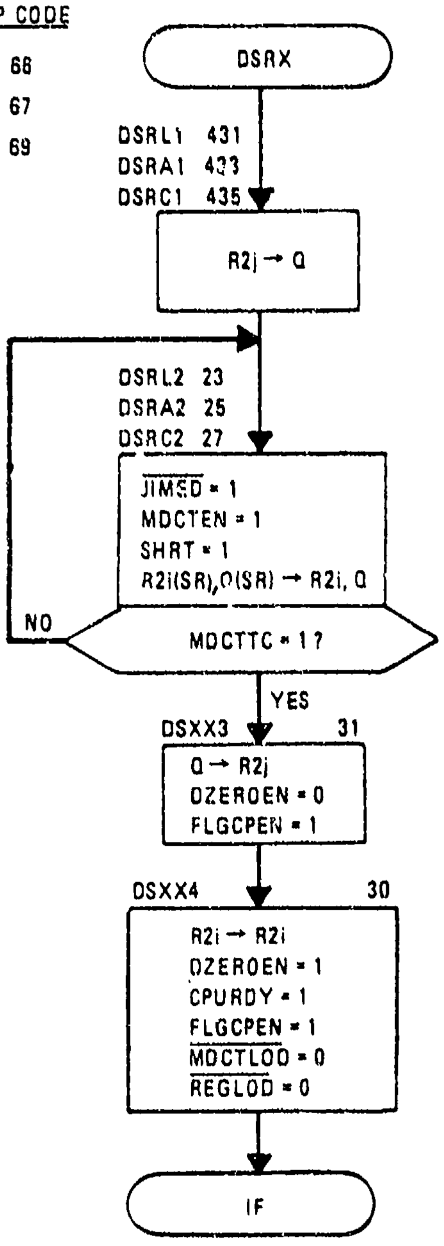


Figure 68. DSRA Timing Diagram

DOUBLE PRECISION SHIFT RIGHT: R2i, R2j → R2i, R2j (SHIFTED N + 1 TIMES)

TYPE	OP CODE
DSRL	66
DSRA	67
DSRC	69



77-0819-VA-46

Figure 69. DSRA Instruction

MNEMONIC: DSRC OP CODE: 69

SHORT NAME: double shift right cyclic

FORMAT: DSRC R2, N

0	1	1	0	1	0	0	1	N-1	R2

DESCRIPTION: The content of registers R2 and R2 + 1 is shifted right cyclically N positions. The field N-1 being zero represents a shift of 1 position. The field N-1 being 15 represents a shift of 16 positions. Bits leaving the least significant position of register P2 + 1 enter the sign position of register R2. Bits leaving the least significant position of register R2 enter the sign position of register R2 + 1. No bits are lost. The condition status, CS, is set based on the double precision result in registers R2 and R2 + 1. R2 must be even.

Result in Registers	Resulting Condition Status		
R2, R2 + 1	Bits	Hex	JC Mnemonic
both zero	0010	2	EZ
sign bit of R2 = 1	0001	1	LZ
otherwise	0100	4	GZ

REGISTERS AFFECTED: R2, R2 + 1, CS

TIMING: 1.8 + 0.4 per position

TYPE - R (REGISTER TO REGISTER INSTRUCTION)

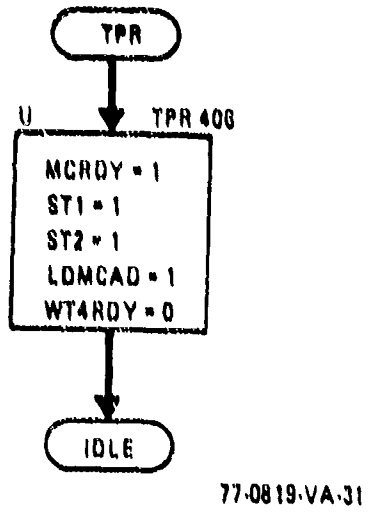


Figure 70. Type - R (Register to Register Instruction)

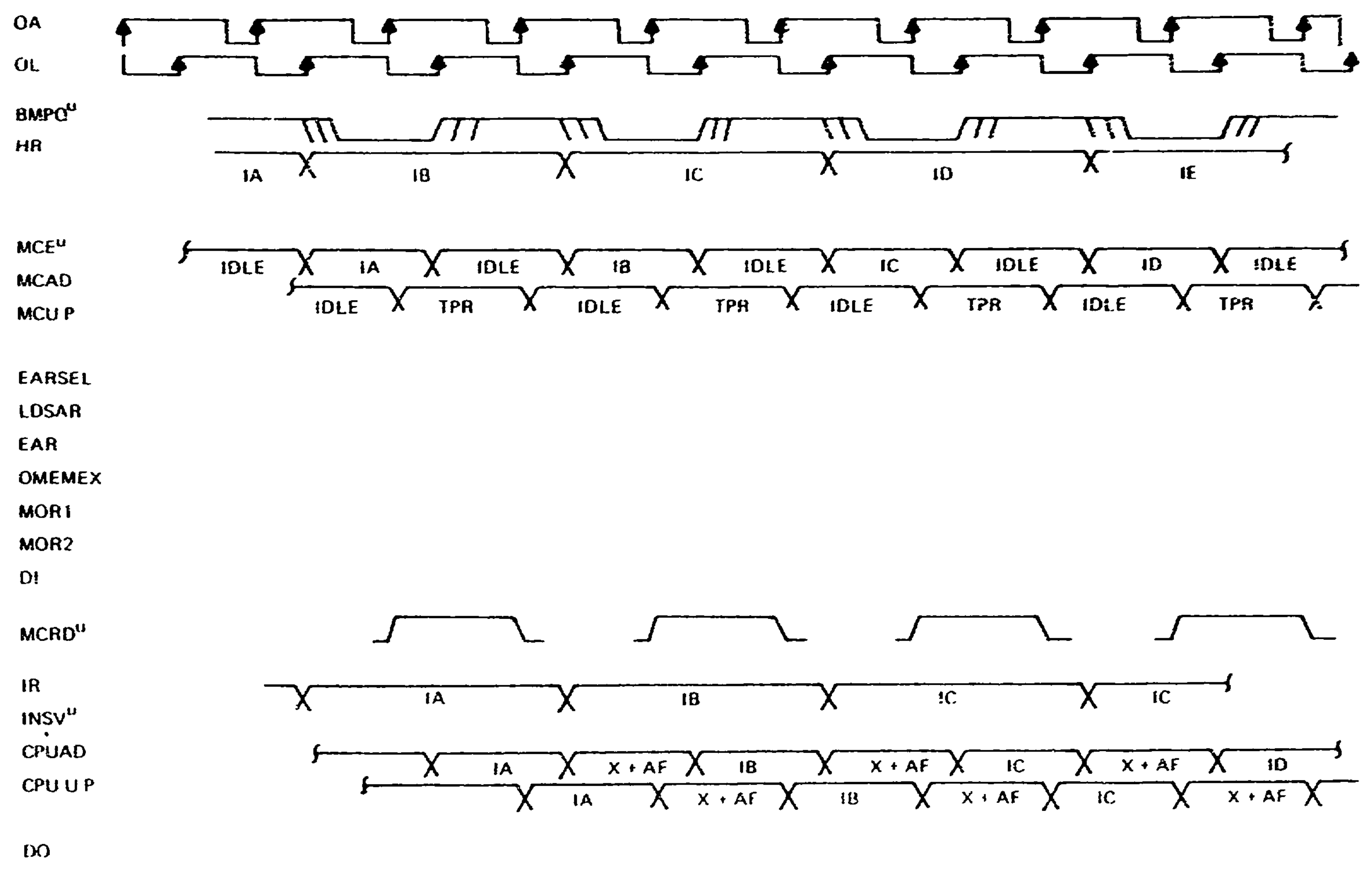
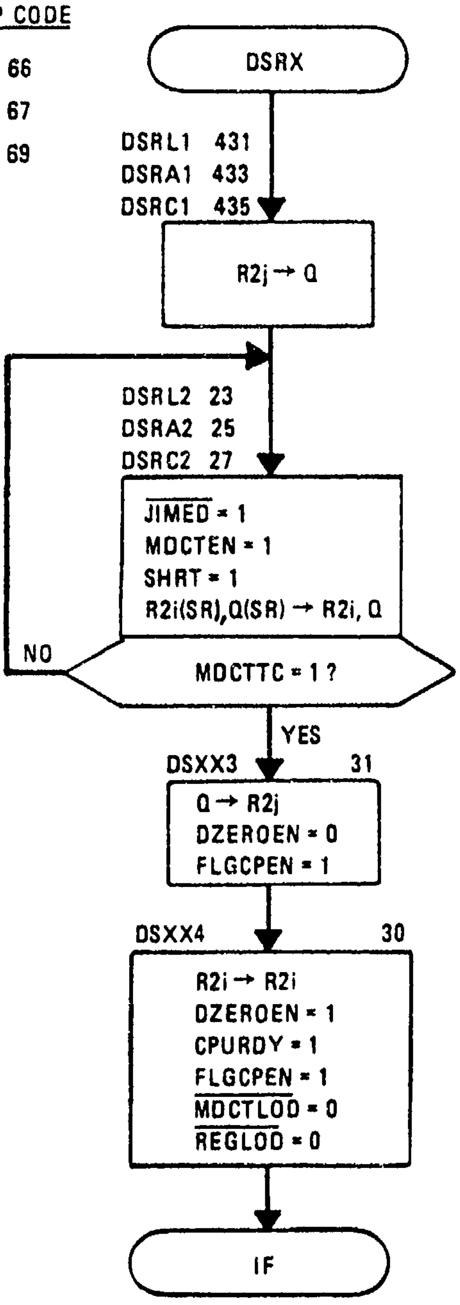


Figure 71. DSCR Timing Diagram

DOUBLE PRECISION SHIFT RIGHT: R2i, R2j → R2i, R2j (SHIFTED N + 1 TIMES)

TYPE	OP CODE
DSRL	66
DSRA	67
DSRC	69



77-0819-VA-46

Figure 72. DSCR Instruction

SECTION IV

LOW-LEVEL MACHINE (LLM) DESIGN

4.1 SCOPE OF DESIGN

The results of the software analysis performed during this contract provided the natural foundation for a family of airborne digital computers. With appropriate modifications, the present AYK-15 computer would become the high performance member of the computer family. However, the "low end" or lower performance members of the family were yet to be defined. It is felt by both the Air Force and Westinghouse, that this "Low Level" machine should be instruction set compatible with the higher members of the family, while minimizing cost, power and volume; and still using the same support software package and facilities.

To this end, an investigation and block level design was performed to more fully define the characteristics of this Low-Level Machine (LLM). This investigation resulted in a detailed study of machine architectures suitable for the LLM implementation as well as an I/O interconnect definition (I-BUS) amenable to I/O expansion and CPU interconnection (multiprocessing). The results of this investigation are part count, power and execution time estimates for the proposed LLM.

What follows is a summary of this investigation which concludes with a block level description of the proposed LLM.

4.2 APPLICATION BASE OF LLM

The first step in the LLM investigation was to define the type of problem to be solved by the LLM. Since the computer is intended to be used in a multitude of applications, an application base had to be defined for the new machine in order to limit the scope of the investigation. With the help and experience of AFAL, it was decided that the LLM should be used primarily in a multicomputer avionics environment. It would, therefore, perform pre-processing of sensor data prior to transmission of the data to other processors within the system. Similarly, the LLM would perform any post processing necessary for actuator data. Figure 73 illustrates a desired application environment for the LLM.

Since the sensor/actuator requirements may be quite diverse from one aircraft to another, the LLM should also provide an efficient means of interconnection of groups of LLM's to modularly expand the data handling capability of the sensor/actuator system. Therefore, as the number of sensors for the system increases, additional LLM's may be added in a modular building block fashion as illustrated in figure 73.

Using this application model as a starting point, past programs were reviewed by AFAL and Westinghouse in order to establish the throughput required for the LLM. With the throughput defined, a set of design goals were then established for the LLM.

4.3 DESIGN GOALS

A set of five design goals were established to provide guidelines for the LLM design. They were:

a. Upward software campatability with DAIS (AYK-15)

b. 2.5 to 5.0 μsec 16-bit fixed-point ADD

c. Universal memory interface

d. I-BUS I/O design

e. Minimize volume and power

Software compatability with the modified AYK-15 machine, was given the highest priority as a design goal in order to take advantage of the software support developed for the AYK-15. However, wherever necessary, instructions were omitted from the LLM to simplify its structure and minimize the parts count. As a result, the LLM became "upward compatable" with modified AYK-15 computer. (See Paragraph 2.5.3, Subset for LLM).

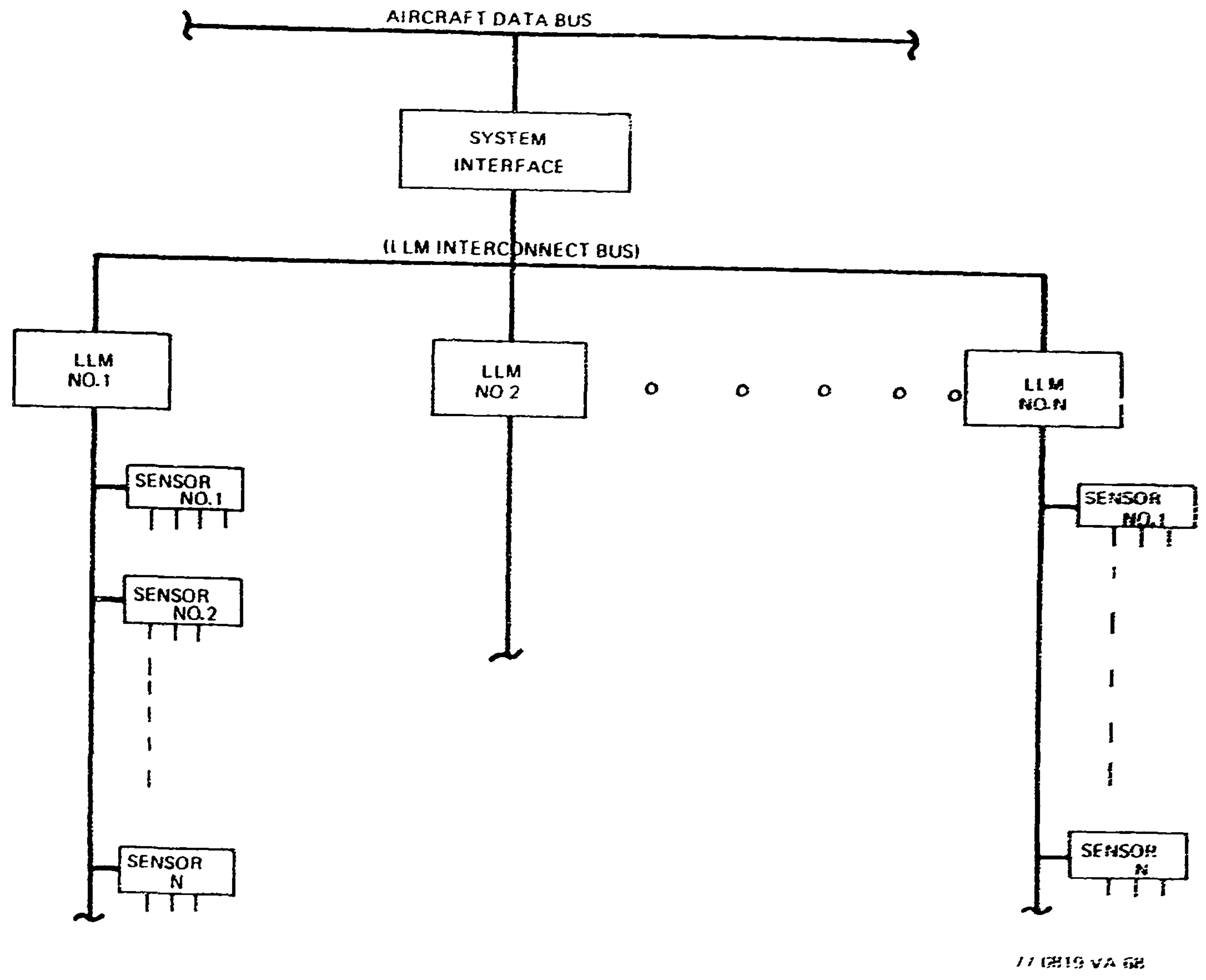


Figure 73. LLM as a Preprocessor

After reviewing the application bases for the LLM, the goal of 2.5 μsec was deemed suitable for a 16-bit fixed-point ADD execution time. This design goal permitted a sizing of the control portion of the LLM to provide a starting point for the design effort.

Because the LLM was intended to be used across a wide range of applications, it was felt that the ability to adapt the LLM to a particular application by varying the memory organization was highly attractive. Therefore, a generalized memory interface to allow for varying memory speeds and technologies (IC or Core) was included as a design goal.

In order to provide for modular growth of the I/O and a link for multiprocessor application structures, the I-BUS approach developed by AFAL (Final Report, Cont. No. F33615-74-C-1018) was adopted as the standard I/O interface.

Finally, in order to reach a maximum application base it was deemed desirable to minimize the volume of the LLM by use of available LSI technology wherever practical. To this end, speed and performance were sacrificed, within the established design goals, to allow for a minimum parts count (and hence volume) configuration.

4.4 LLM ORGANIZATION

4.4.1 Arithmetic Loop

The DAIS instruction set is organized around a general register machine utilizing a group of 16 general registers. This, along with the desired speed goals dictated the choice of the AM-2901 μ-processor as the building block of the LLM arithmetic unit. Figure 74 illustrates the resulting architecture for the LLM.

The LLM is organized around a single 16-bit data bus (MDTA) within the CPU. Memory, I/O and CPU data are all transferred over this bus. Two groups of 8-bit wide 2901's are used to process data and form the register file for the LLM. Registers, MOR1 and MOR2 are memory operand registers used as intermediate buffer registers. SCT is a 5-bit

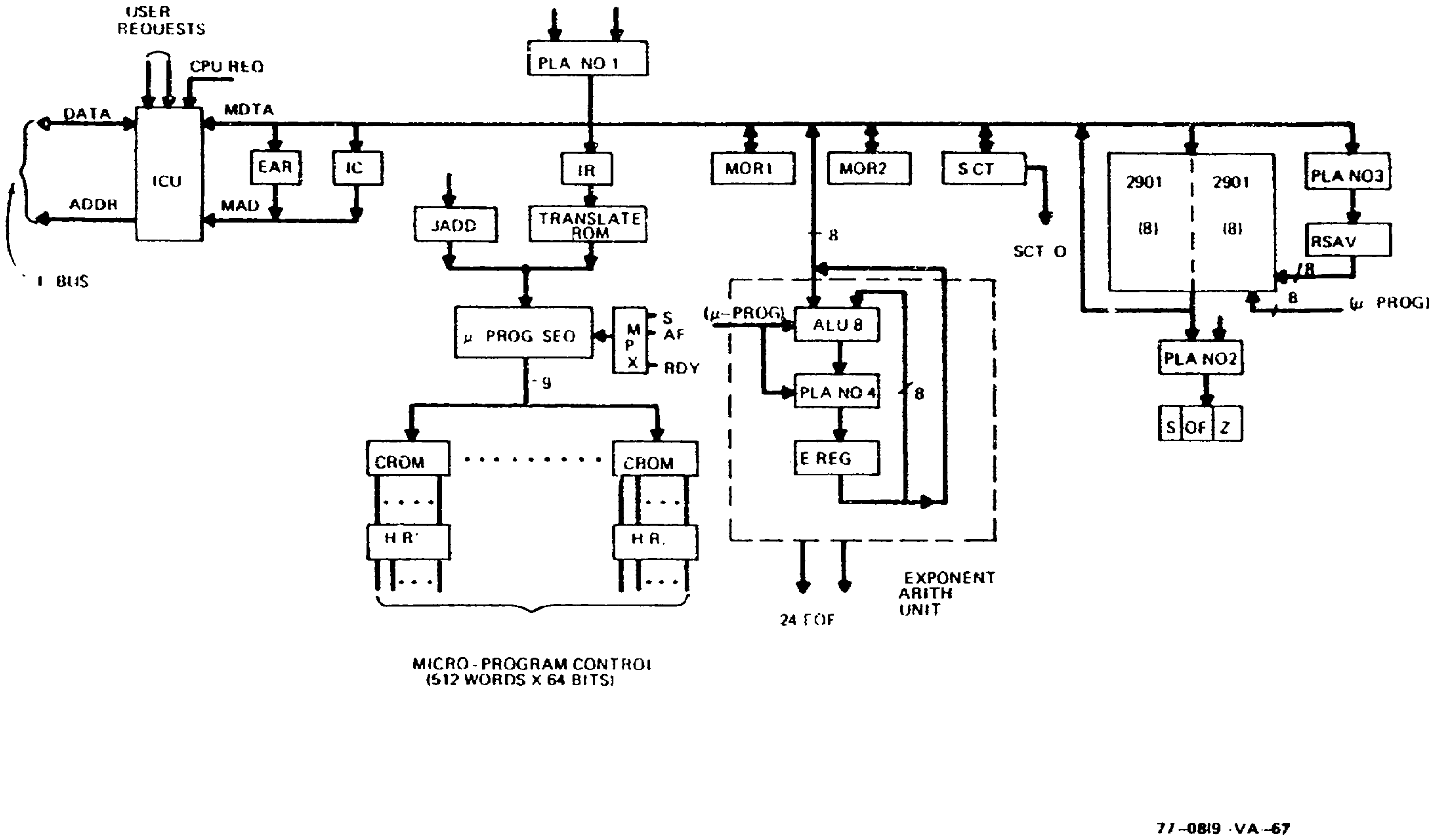


Figure 74. LLM CPU Organization

counter register used as a sequence counter for multiple clock microprogram routines.

4.4.2 Control Structure

The control portion of the LLM is comprised of a 512-word by 64-bit microprogram store contained in Read Only Memories (ROM's). A microprogram sequencer (such as the AM-2911) is used to control the sequencing of the microprogram instructions for CPU algorithm execution. Microprogram address sources may be selected from either a microprogram jump field (JADD ROM) or from a set of ROM's to allow efficient microprogram branch capability. Also, system flags may be individually tested by the microprogram sequencer to facilitate conditional microprogram branching.

Each microprogram ROM output is followed by a holding register to allow microinstruction fetches to be overlapped with microinstruction execution.

Discrete registers are provided for the formation of the effective address (EAR) for memory address instructions and for the instruction counter (IC). Each of these registers and the MDTA bus are connected to the I-BUS Control Unit (ICU) which provides the interface to the I-BUS. The memories and I/O are then interfaced to the I-BUS.

4.4.3 I/O Organization

The I/O and memory system is interfaced with the I-BUS to provide a standard interface for all I/O elements. Therefore, a standard set of I/O modules may be developed and a LLM application configuration by simply "plugging in" the appropriate modules. An I/O module may be as simple as a discrete interface or as complex as a 1553-A processor (figure 75).

The memory system is interfaced similarly to an I/O device, through the MIU (Memory Interface Unit). Any memory technology (IC, Core,

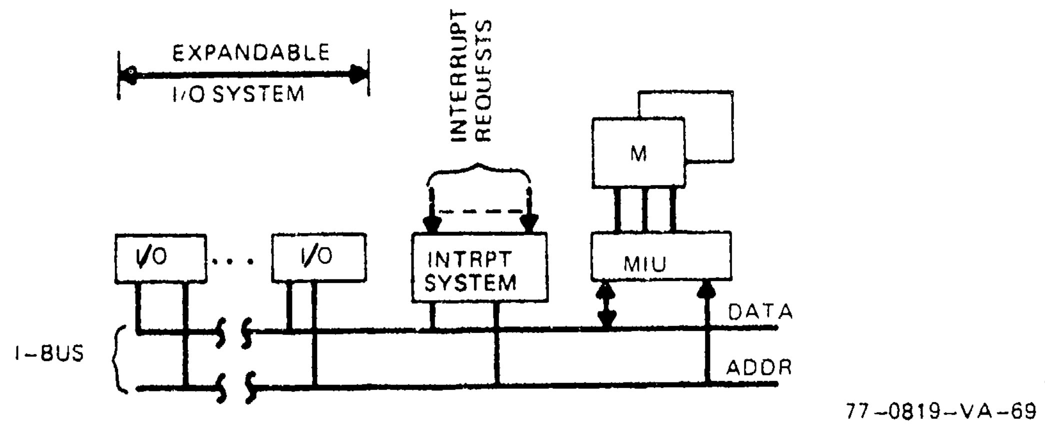


Figure 75. LLM I/O Organization

CCD, etc.) may be interfaced with the MIU since all memory timing is performed in a "handshake" fashion.

The interrupt system is interfaced directly with the I-BUS and provides sixteen levels of priority interrupts to the CPU.

4.4.4 Machine Operation and Timing

In order to more fully understand the operation of the LLM, five microprogram control routines will be described in detail. The routines are:

a. Instruction Fetch
b. Fixed point ADD (Register/Memory)
c. SHIFT Instructions
d. Floating point ADD
e. Multiply instruction

A microprogram flow chart is included for each of these instructions to facilitate the explanation.

a. Instruction Fetch

During the Instruction Fetch cycle, the CPU reads the current 2-word instruction to be executed and saves it in IR, MOR1 and MOR2. Referring to figure 76, each step of the microprogram execution for the Instruction Fetch cycle is indicated as a separate block. Figure 77 provides the detailed timing for the Instruction Fetch cycle.

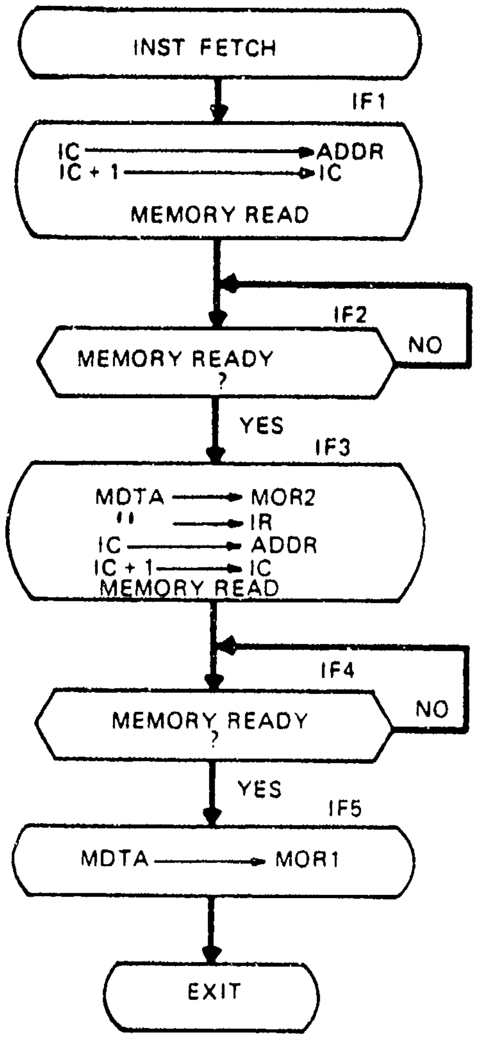
The Instruction Fetch begins by passing the IC to the ICU and requesting a memory read operation from the memory system (IF1 of Figure 76). The CPU Control then increments the IC and proceeds to Step IF2 to await the completion of the memory cycle. When the memory data is ready, the CPU proceeds to IF3 and loads the fetched memory word (most significant 16 bits of the 32-bit instruction word) into IR and MOR2. A new memory cycle is then initiated to read the second half of the instruction. Once again, the IC is incremented and the CPU waits for the completion of the memory cycle. When the memory cycle has ended, the CPU proceeds to step IF5 and loads register MOR1 with the second half of the instruction.

The instruction fetch cycle is then completed with the instruction saved in MOR1 and MOR2. The CPU next proceeds to execute the instruction before returning to the Instruction Fetch cycle. Figure 77 illustrates the detailed timing for this sequence of events.

b. Fixed-Point ADD

The Fixed-Point ADD performs a parallel 16-bit two's complement ADD of an accumulator register (RA) and a memory operand. The sum is placed in RA and the appropriate arithmetic flags are sampled.

Referring to figures 78 and 79, the CPU begins execution of the ADD instruction by calling a micro-program subroutine to compute the effective address of the memory operand. The subroutine returns the calculated address in the EAR register.



77-0819-VA-70

Figure 76. Instruction Fetch Flow

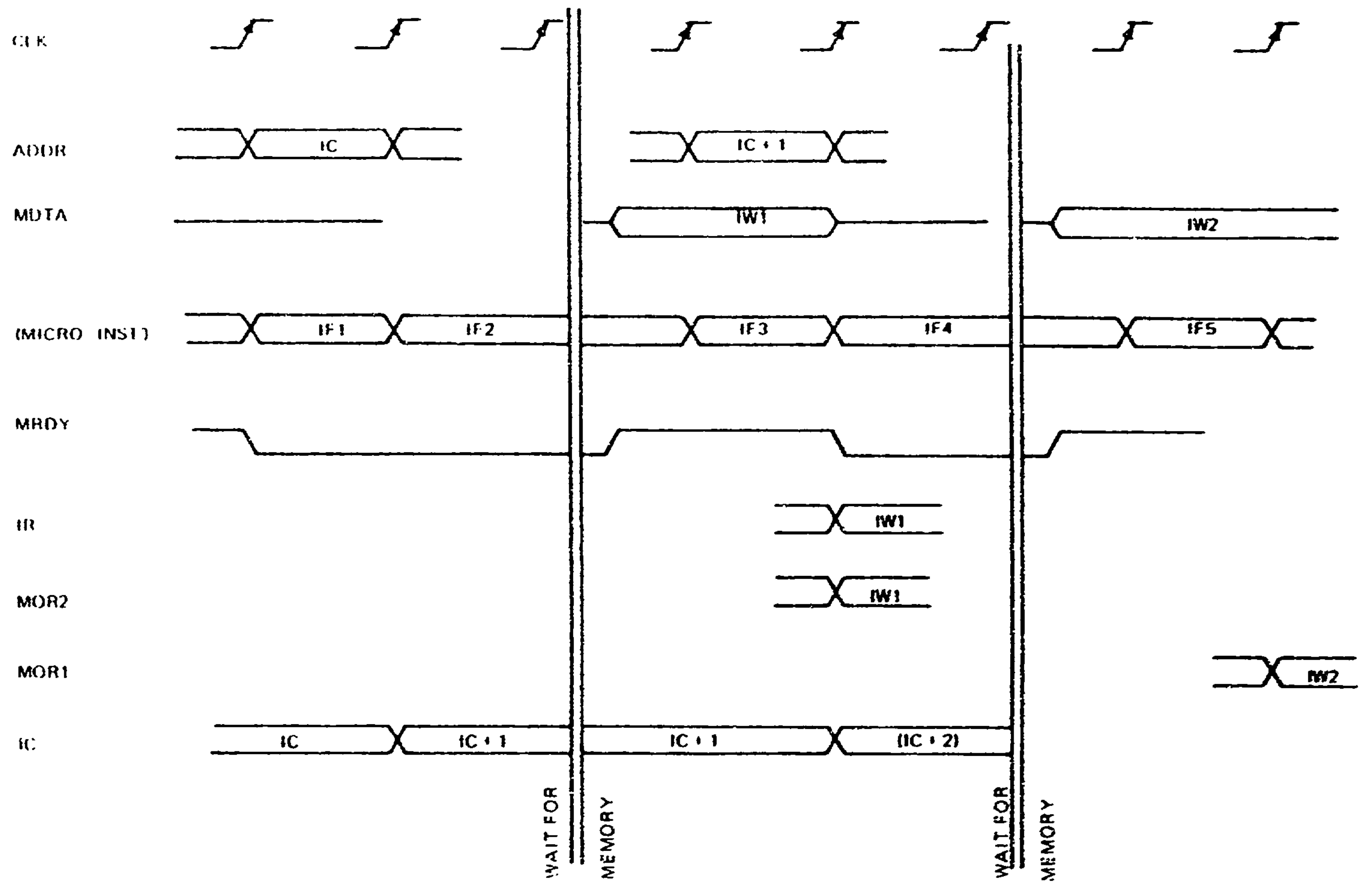
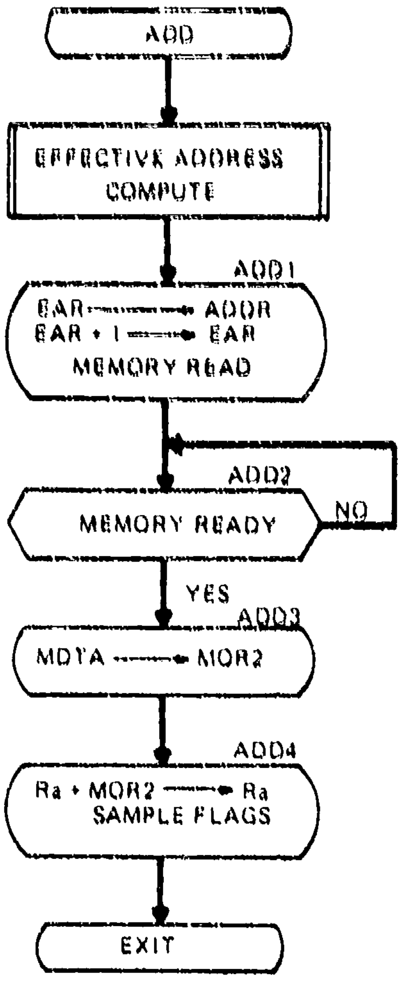


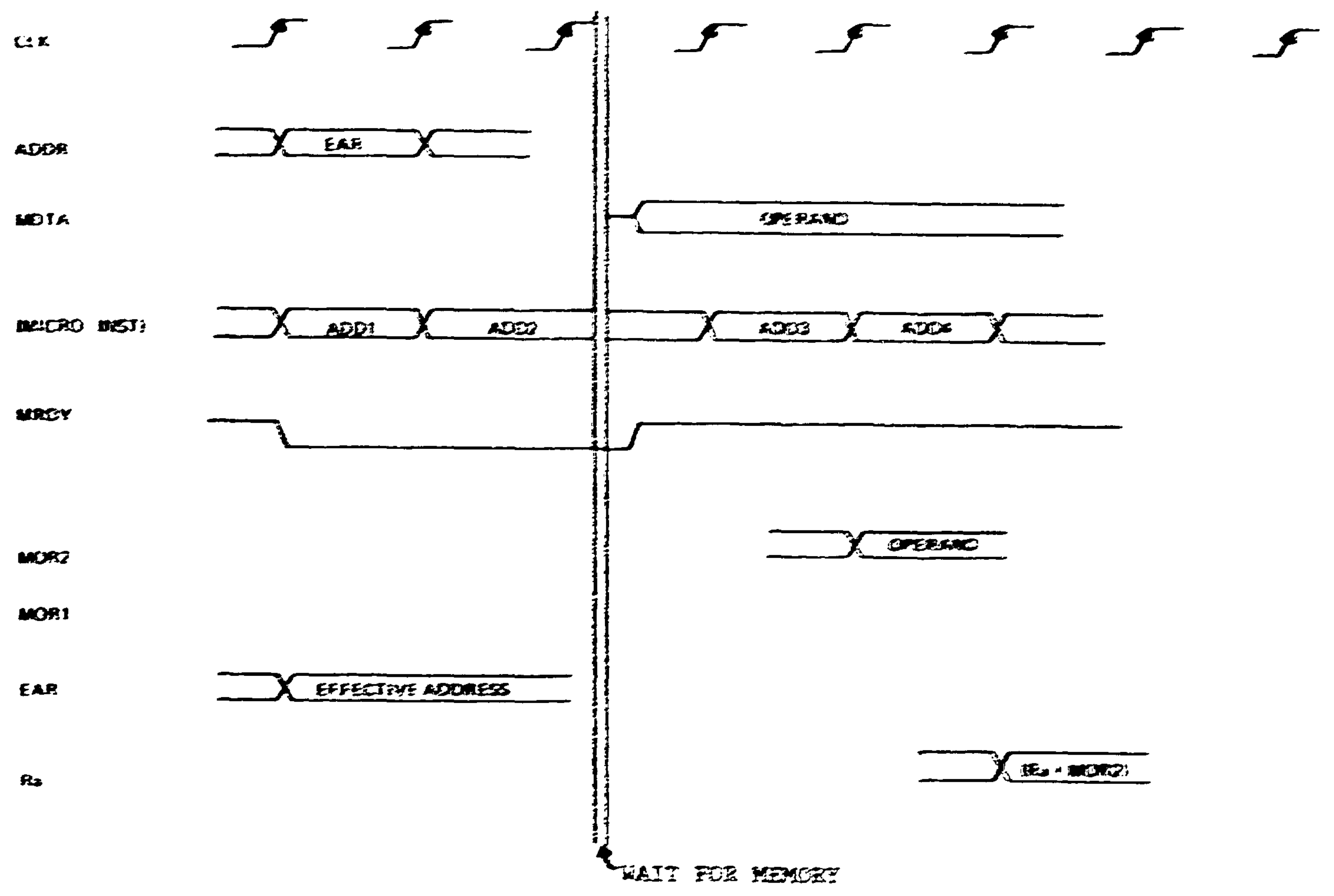
Figure 77. Instruction Fetch Timing

77 0819 VA 71



77-0819-VA-72

Figure 78. Fixed-Point ADD Flow



77 0839 VA 72

Figure 79. Fixed-Point ADD Timing

During step ADD1, the effective address is passed to the memory and a memory read is initiated. The CPU waits for the memory to complete its read cycle in ADD2. When completed, the CPU loads the memory data into MOR2 at Step ADD3.

The CPU now has both operands for the fixed-point ADD and completes the ADD operation during Step ADD4. During ADD4, MOR2 is enabled onto the MDTA bus and passed to the 2901 μ-processor. The CPU control ROM's instruct the microprocessor to perform a 16-bit fixed-point ADD to RA and return the result to RA. Simultaneously with RA being loaded with the sum, the three arithmetic flags (<u>S</u>ign, <u>O</u>verflow, <u>Z</u>ero) are updated to reflect the results of the arithmetic operation.

The CPU has now completed the ADD instruction and returns to initiate the next instruction fetch cycle.

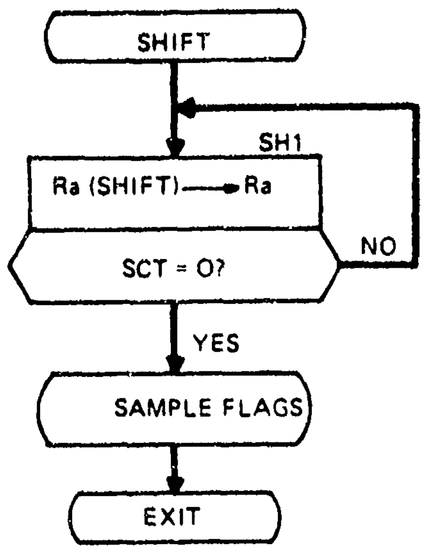
c. SHIFT Instruction

Figures 80 and 81 illustrate the execution of the SHIFT instruction. During SH1 register R_A is repeatedly shifted while SCT (which contains the shift count) is decremented. The microprogram sequencer continually tests the value of SCT and causes microprogram control to be passed to step SH2 when SCT is zero. During SH2, the arithmetic flags are sampled and finally the next instruction fetch cycle is begun.

d. Floating Point ADD

The floating point instruction performs a 32-bit floating-point ADD (8-bit exponent and 24-bit fractional mantissa) between the double register pair (R_A, R_{A+1}) and the double-memory word designated as the operand. The result is returned to (R_A, R_{A+1}) replacing one of the original operands. Both operands are assumed to be normalized floating point numbers and their sum is normalized prior to placement in (R_A, R_{A+1}).

For purposes of discussion let R_E represent the exponent portion of the register operand while M_E represents the exponent portion of the memory operand.



77-0819-VA-73

Figure 80. Shift Instruction Flow

Referring to figure 82, the algorithm begins with an effective address calculation for the memory operand. The double-word memory operand is then read from memory and the most significant half saved in MOR2 while the least significant half is saved in MOR1. The floating-point algorithm now being with microprogram step FPA1.

During FPA1, R_E (exponent field of the register operand) is transferred into EREG of the Exponent Arithmetic Unit (see figure 74). The next microprogram step performs an "excess 128" subtract in the exponent arithmetic unit forming $(R_E - M_E)$. This represents the exponent difference (ΔEXP) of the two numbers and will be used to indicate which operand needs to be adjusted (shifted right).

The operand adjustment algorithm begins at FPA3 where the sign of ΔEXP is tested to determine which operand is to be adjusted. Assuming that $R_E \geq M_E$, the control proceeds to FPA4.

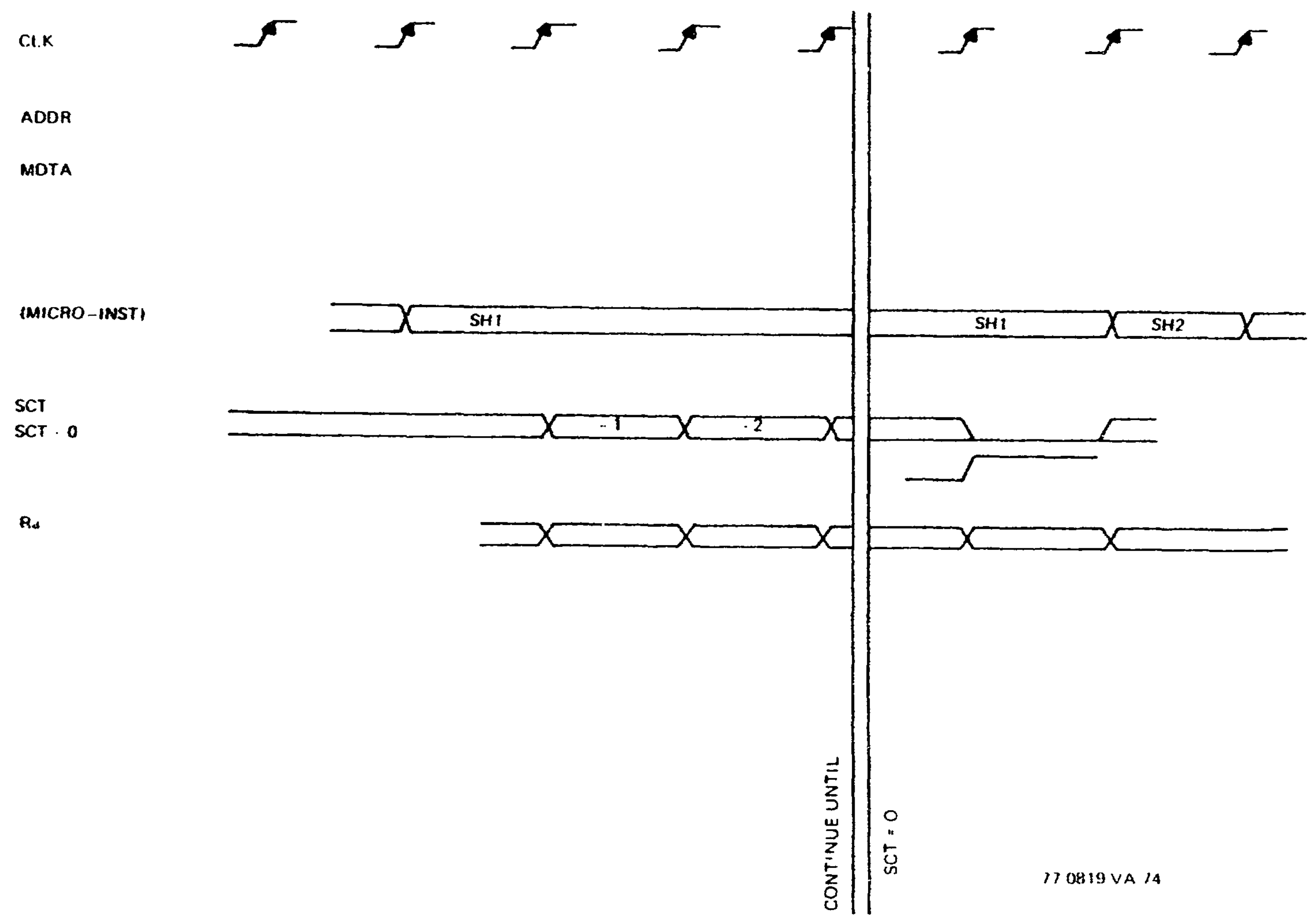


Figure 81. Shift Timing

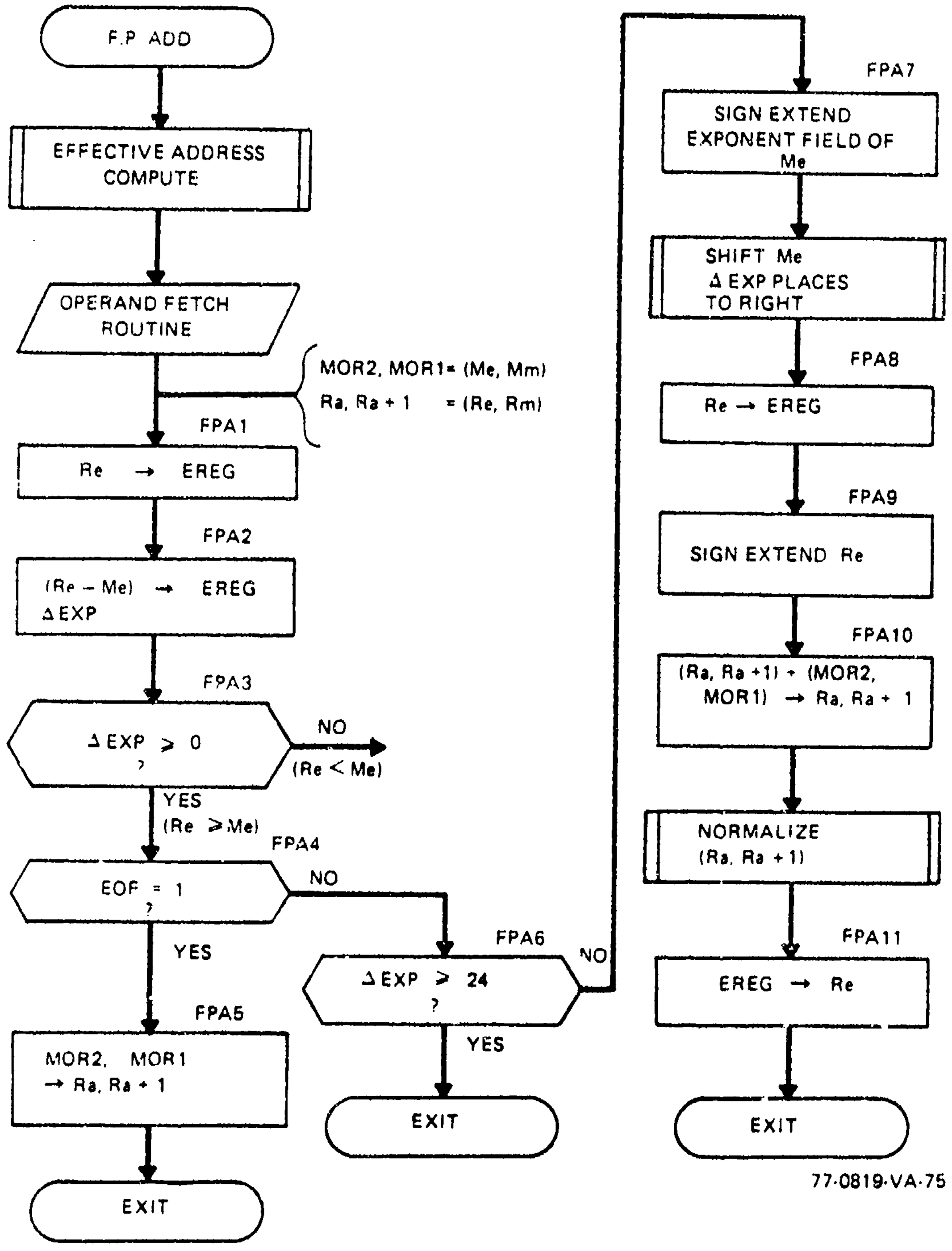


Figure 82. Floating - Point ADD

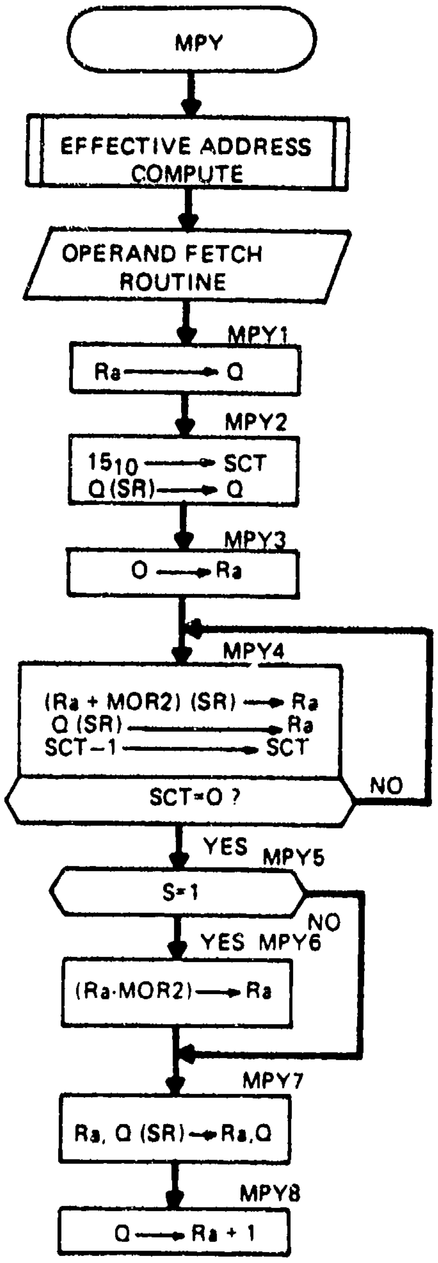
If the exponent differencing did not overflow then the microprogram proceeds to FPA6 where it tests to see if the memory operand may be successfully scaled. If the ΔEXP value in EREG is greater than or equal to 24, then no further calculations need be performed and the register operand will be the answer. However, if the memory operand can be successfully scaled, the microprogram proceeds to FPA7 where PLA#1 is used to sign extend the mantissa through the exponent field of the memory operand in MOR2. Next, the register pair (MOR2, MOR1) is shifted right ΔEXP places in a microprogram subroutine. The memory operand is now appropriately scaled for mantissa addition.

FPA8 loads EREG, with the answer exponent (R_E) and proceeds to FPA9 where the exponent field of (R_A, R_{A+1}) is sign extended in preparation for the mantissa add operation of step FPA10. After the mantissas are added, microprogram control is passed to a normalize subroutine where the answer mantissa is shifted left until it is appropriately normalized. Of course, with each shift left required for normalization, the answer exponent in EREG is decremented. Upon completion of the normalization subroutine, the answer exponent in EREG is assembled into R_E and the instruction is complete.

e. Multiply

The fixed-point multiply is performed entirely within the 2901 microprocessor using a one bit at a time repeated add algorithm.

Referring to figure 83, the multiply algorithm begins with an effective address computation followed by an operand fetch for the multiplicand. The multiplication "setup" begins with step MPY1 by transferring the multiplier to the Q register within the 2901 microprocessor. MPY2 loads the constant 15_{10} from PLA#1 (see figure 74) into SCT and shifts Q one place right entering the least significant multiplier bit into the S flip flop. Next, R_A is cleared during MPY2 to act as the partial sum register for the multiply.



77–0819–VA–76

Figure 83. Multiply Flow

The repeated sums are performed during step MPY4 using the S flip-flop to control the add operation within the 2901. As each sum is formed the result and multiplier are shifted one place right to form the next partial sum. The process now continues until 15 partial sums are formed at which time control is transferred to MPY5.

In accordance with the rules for performing two's complement multiplication, MPY5 tests the sign of the multiplier to determine if a correction cycle for the partial sum is necessary. If required, MPY6 performs the required subtraction. MPY7 adjusts the partial sum for integer representation while MPY8 moves the least significant half of the product into R_{A+1} to complete the instruction.

4.4.5 Execution Times

Instruction execution times for the LLM are a function of two criteria. First, the memory speed has a direct impact upon both instruction fetch times and operand fetch times. Secondly, the internal circuit delays of the LLM dictate a maximum frequency for the CPU clock. Using a one microsecond core memory for instructions and data with a four megahertz system clock, the following typical instruction times are achievable:

LOAD	3.0 μsec
ADD	3.0 μsec
STORE	3.0 μsec
SHIFT	2.25 + (N-1) 0.25 μsec
MPY	8.5 μsec
FP ADD (average)	10.5 μsec

4.5 PHYSICAL DESCRIPTION

Using the machine organization shown in figure 74, an estimate of parts was made to "size" the LLM. Once a parts estimate was obtained, an estimate of power consumption was then made. For purposes of estimation, the memory parts and power were omitted while the I/O configuration was assumed to be a 16-level priority interrupt system.

Using presently available parts, table 9 reflects the parts estimates for the LLM. Accordingly, the LLM could be fabricated from approximately 120 currently available bipolar devices. Using packaging techniques similar to the present DAIS computer, the LLM would occupy three printed wiring boards and dissipate approximately 45 watts.

TABLE 9

LLM PARTS AND POWER ESTIMATES

ELEMENT	LSI	MSI	SSI	POWER (WATTS)
CPU	19	32	10	30
ICU	9	16	15	10
I/O	2	4	15	5
TOTAL	30	52	40	45

77-0819-VA-77

☆U.S.Government Printing Office: 1978 — 757-080/617